Titre: Title:	Architectures haute performance pour transformations géométriques à trois dimensions
Auteur: Author:	Eric Achard
Date:	1997
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Achard, E. (1997). Architectures haute performance pour transformations géométriques à trois dimensions [Master's thesis, École Polytechnique de Montréal]. PolyPublie. https://publications.polymtl.ca/6693/

UNIVERSITÉ DE MONTRÉAL

ARCHITECTURES HAUTE PERFORMANCE POUR TRANSFORMATIONS GÉOMÉTRIQUES À TROIS DIMENSIONS

ÉRIC ACHARD
DÉPARTEMENT DE GÉNIE ÉLECTRIQUE
ET DE GÉNIE INFORMATIQUE
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION
DU DIPLÔME DE MAÎTRISE ÈS SCIENCES APPLIQUÉES
(GÉNIE ÉLECTRIQUE)
DÉCEMBRE 1997

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé:

ARCHITECTURES HAUTE PERFORMANCE POUR TRANSFORMATIONS GÉOMÉTRIQUES À TROIS DIMENSIONS

présenté par: ACHARD ÉRIC
en vue de l'obtention du diplôme de: Maîtrise ès sciences appliquées
a été dûment accepté par le jury d'examen constitué de:

M. BOIS Guy, Ph.D., président
M. SAVARIA Yvon, Ph.D., membre et directeur de recherche
M. BLAQUIÈRE Yves, Ph.D., membre

Je dédie ce mémoire à ma femme, à mes parents et mon enfant.

Remerciements

Je remercie beaucoup mon directeur Yvon Savaria, qui malgré un horaire très chargé, a toujours su trouver du temps pour me rencontrer, me guider et répondre à mes nombreuses questions.
Je remercie Hakim Khali qui lui aussi n'a pas été avare de son temps afin de m'aider.

De plus, j'aimerais remercier le centre d'excellence IRIS pour son support financier, et le Centre National de Recherche Canadien pour m'avoir fourni la problématique qui a occupé mon temps pendant deux ans.

Résumé

Afin de combler le besoin grandissant pour des caméras à trois dimensions de haute performance, le CNRC (Centre National de Recherche Canadien) a développé une caméra visant la reconstruction de modèle synthétique d'objet réel.

Cette caméra est basée sur le balayage d'un objet par un rayon laser, à l'aide d'un jeu de miroirs. Les données fournies par ce système dépendent des angles des miroirs ainsi que de la position du centroïde du rayon réfléchi sur un détecteur. L'algorithme qui permet de transposer ces données en des points de l'espace cartésien est basé sur des opérations arithmétiques et des fonctions trigonométriques. La version de la caméra qui est présentement en développement, vise une performance de 1 à 10 millions de points par seconde. Afin que l'algorithme de transformation puisse suivre la cadence élevée de la caméra, ce projet étudie des méthodes d'accélération de calculs et différents systèmes susceptibles de rencontrer les performances visées.

Trois types de systèmes ont été étudiés en détail : logiciel, matériel et mixte. En premier lieu, une architecture mixte (logicielle-matérielle) a été conçue. Ce système est basé sur trois processeurs TMS320C40 de Texas Instruments ainsi que sur un FPGA de Xilinx. Les performances obtenues ont atteint le but visé, mais dû aux communications entre les processeurs, elles sont moins grandes que celles que nous laissait espérer la puissance de calcul de trois processeurs et le coût de ce système.

La version logicielle a été conçue afin d'être implantée sur deux processeurs : le TMS320C40 et le ADSP-21060 de Analog Devices (SHARC). Grâce aux unités arithmétiques performantes et à la grande quantité de mémoire interne du SHARC, les performances visées ont été atteintes facilement, tandis que le TMS320C40 n'a pas fait mieux que 60% de notre objectif.

La version matérielle a été développée afin d'être implantée sur un FPGA de Xilinx. Ce système, comme nous pouvions nous y attendre, puisque c'est un circuit dédié, est celui qui a fourni les meilleures performances, mais à un coût de développement plus élevé.

Abstract

In order to satisfy the growing demand for fast 3-D vision applications, the NRC (National Research Council) developed an optical system for synthetic reconstruction of real objects. This optical system scans an object using a laser beam and rotating mirrors. The data provided by the system depend on the angles of the mirrors and the position of the reflected beam on a detector. The algorithm used to remap the raw data into points in the cartesian coordinate system is based on arithmetic operations and trigonometric functions. The optical system under development aims at a performance between 1 and 10 million points per second. In order for the algorithm to compute the data in real time, this project looks at ways to accelerate the computation, and studies systems capable of meeting these performances. Three types of implementations have been studied : software, hardware and mixed.

First, a mixed architecture was designed. This system is based on three processors TMS320C40 from Texas Instruments and on a FPGA from Xilinx. The performance goal was achieved, even though the end results were lower than expected, considering the system's cost. This disappointing outcome was due to the communications between the processors.

The software version has been developed to fit on two different processors : the TMS320C40 and the ADSP-21060 from Analog Devices (SHARC). Because of the high performance of the arithmetic units and the large amount of internal memory of the SHARC, the desired performances were easily reached, whereas the TMS320C40 achieved only 60% of our goal.

Finally, the hardware version has been designed for a FPGA implementation on a Xilinx. This system, because its architecture is specific to the application, gave the best results, but it required more development efforts than the software version.

Table des matières

Liste des figures

Liste des tableaux

Liste des abbréviations

CLB : "Configurable Logic Block"

CNRC : Centre National de Recherche Canadien

CPU : "Central Processing Unit"

DMA : "Direct Memory Access"

DRAM : "Dynamic Random Aceess Memory"

DSP : "Digital Signal Processing"

FIFO : "First-in First-Out"

FPGA : "Field Programmable Gate Array"

FPU : "Floating Point Unit"

FXU : "Fixed Point Unit"

LRU : "Least Recently Used"

LSB : "Least Significant Bit"

LUT : "Look-Up Table"

MSB : "Most Significant Bit"

SHARC : "Super Harvard Architecture Computer"

SRAM : "Static Random Access Memory"

UAL : Unité Arithmétique et Logique

Liste des annexes

Chapitre 1 : Introduction

De nombreux domaines tels que le militaire, le médical et l'aérospatiale, ont poussé le développement de l'imagerie à trois dimensions. Qu'on pense à la détection de faux objets d'art ou à la reconnaissance de formes par satellite, le besoin constant de l'augmentation de la résolution et de la vitesse des caméras 3-D représente un défi de taille pour le monde de la vision numérique. Plusieurs méthodes de reconstitution de scène 3-D sont présentement utilisées. Chacune de ces méthodes a des particularités qui la relie généralement à des besoins bien spécifiques.

La caméra qui est étudiée dans ce mémoire vise la reconstruction de modèles synthétiques d'objets réels. Elle utilise le principe de triangulation active par balayage auto-synchronisée développé par le CNRC (Conseil National de Recherche Canadien). Cette technique est basée sur le balayage d'un objet par un rayon laser grâce à un jeu de miroirs rotatifs, et la récupération de la réflexion du rayon réfléchi par l'objet à l'aide d'un capteur. Grâce aux angles des miroirs et à la position de la centroïde sur le capteur, il est possible de déterminer la coordonnée d'un point dans l'espace cartésien.

Une première version de cette caméra a été développée par le CNRC. Depuis sa création, son champ d'application est devenu très vaste. Par exemple, une version à haute précision a permis d'immortaliser des objets de musée afin de prouver leur authenticité. De plus, elle peut être utilisée à des fins de "reverse-engineering". En effet, une pièce peut être reproduite de façon très précise grâce aux données fournies par la caméra. Une autre version de la caméra nécessitant une moins grande résolution que la première mais exigeant une plus

grande vitesse, a été conçue pour permettre l'inspection en temps réel sur une chaîne de montage. Finalement une dernière version à accès aléatoire avec un champ de vision très large a été conçue pour des applications aérospatiales.

Cette caméra n'ayant qu'un débit maximal de 20,000 points par seconde, une nouvelle génération de caméra est présentement en développement. Ce nouveau système de triangulation optique (étudié au chapitre 2 de ce travail), vise un débit de 1 à 10 millions de points 3-D par seconde.

La transformation des données fournies par la caméra en des points du plan cartésien utilise un grand nombre de fonctions trigonométriques. Lors de l'implantation, ces fonctions sont approximées par des séries qui demandent un grand nombre d'opérations arithmétiques. Le CNRC a donc développé un algorithme qui fait l'approximation de l'algorithme original afin de diminuer le nombre de calcul de fonctions trigonométriques. Malgré cette diminution de complexité, le nombre d'opérations à effectuer et le débit de données très élevé ne permettent pas à des systèmes de calcul conventionnel d'atteindre les performances désirées.

Ce travail vise à trouver des architectures basées sur des processeurs à la fine pointe de la technologie et des circuits programmables susceptibles de pouvoir transformer les données fournies par la caméra en temps réel. Le chapitre 3 examine donc les algorithmes de transformation, des méthodes d'accélération de calcul pour les fonctions trigonométriques, ainsi que la taille et le format des opérandes (arithmétique à virgule fixe versus virgule flottante).

Il existe un très grand nombre d'architectures sur lesquelles il serait possible d'implanter l'algorithme. Le chapitre 4 montre en détail celles qui ont été étudiées lors de ce projet. La

première solution qui a été développée fait partie du groupe d'architectures mixtes matérielle-logicielle. L'architecture de ce système exploite trois processeurs interconnectés qui calculent différentes parties de l'algorithme, dont les données sont fournies par un FPGA. Ce circuit programmable joue le rôle de générateur d'adresses et d'interface avec la mémoire.

La deuxième implantation est une version totalement logicielle. Cette version a été implantée sur deux processeurs : un TMS320C40 de Texas Instruments, et le ADSP-21060 de Analog Devices. Pour les deux processeurs, l'algorithme a été optimisé au niveau du langage assembleur afin d'utiliser le maximum des ressources disponibles sur chacun d'eux, et ainsi d'obtenir les meilleures performances possibles.

Finalement, la dernière solution étudiée est une version totalement matérielle. Une architecture a été développée afin que l'algorithme soit implanté sur un FPGA.

Afin de faire ressortir laquelle de ces architectures est la plus intéressante, le chapitre 5 fait une comparaison entre elles. Cette comparaison prend bien sûr en compte les performances de chacun des systèmes, mais elle considère en plus les coûts pour permettre de comparer la relation coût-performance.

Chapitre 2 : Caméra

Ce travail étant principalement basé sur la caméra 3-D à balayage auto-synchronisé développé par les chercheurs du CNRC, ce chapitre fait une brève revue des plus importants concepts de triangulation qui ont amené le développement de ce système optique.

2.1 Triangulation longitudinale

La figure 2.1 nous montre les trois principales méthodes de triangulation. Elle met en évidence les perturbations influençant le senseur de position. Ce senseur est en fait une matrice d'éléments photosensibles qui récupère la réflexion du rayon laser sur l'objet ciblé.

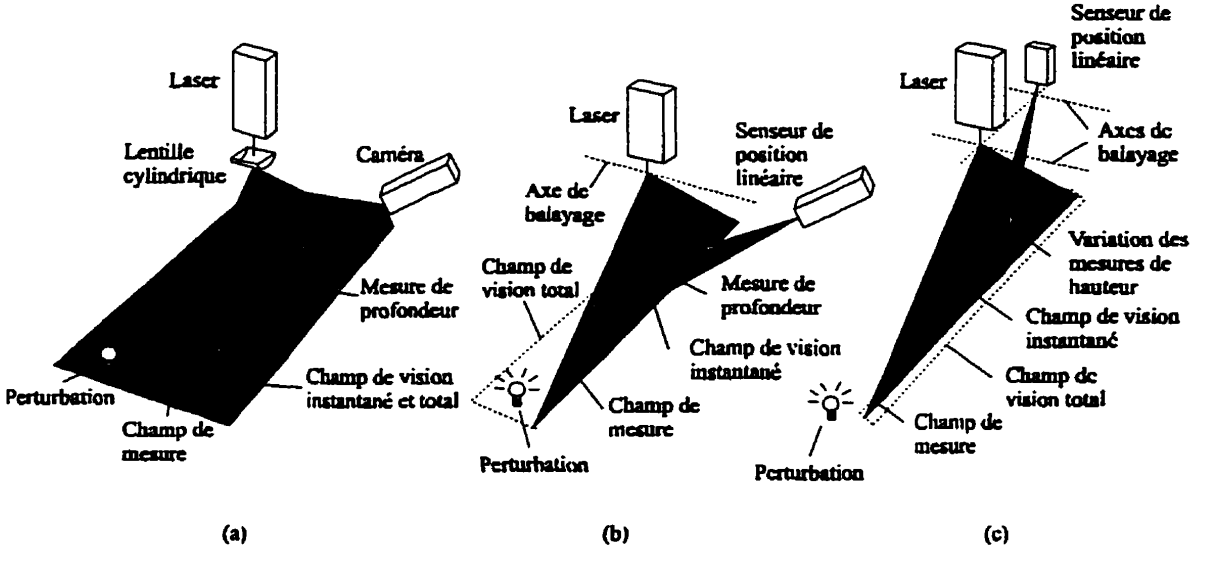


Figure 2.1 : Comparaison du champ de vision instantané de trois approches de triangulation utilisant des sources lumineuses actives : (a) technique avec plan de lumière; (b) méthode de synchronisation latérale; (c) technique de synchronisation longitudinale.

Le système de la figure 2.1a utilise un "plan de lumière", ce qui a l'avantage d'être asynchrone et de fournir un profil 3D au lieu d'un seul point. Le problème de la technique de triangulation qui utilise un plan de lumière provient du fait que les champs de vision instantané et total sont confondus, ce qui rend le système sensible aux perturbations ambiantes. De plus, pour obtenir l'information 3D, un traitement algorithmique sur les données est nécessaire.

La technique de triangulation utilisant la synchronisation latérale est elle aussi sensible aux perturbations à proximité de la zone à échantillonner. La figure 2.1b montre que le champ de vision instantané est relativement étroit, mais que le champ de vision total a néanmoins une dimension considérable. L'avantage de cette technique par rapport à la précédente, est sa résolution spatiale et sa précision.

Finalement, la technique de synchronisation longitudinale (voir fig. 2.1c) restreint le champ de vision total à une dimension du même ordre que celle du champ de mesure. Ce modèle permet une très grande immunité aux bruits, et en plus, elle bénéficie des mêmes avantages que la méthode par synchronisation latérale. La technique de synchronisation longitudinale a donc un avantage marqué par rapport aux deux autres méthodes que nous venons d'étudier. Par contre, pour obtenir une bonne précision, il faut que le laser et le senseur de position soient en synchronisation parfaite. Pour aligner le champ de vision avec le champ de mesure, le CNRC a développé un modèle de triangulation auto-synchronisé (voir section 2.4 : Auto-synchronisation).

2.2 Triangulation active conventionnelle

Le principe de triangulation active (voir fig. 2.2a) est basé sur le balayage de l'objet ciblé par un rayon lumineux généré par un laser. Le balayage proprement dit est produit par un miroir rotatif. La caméra est composée d'une lentille et d'un détecteur photosensible. Elle mesure la location du point lumineux sur l'objet. À l'aide de l'angle du miroir rotatif ainsi que la position du pixel gagnant, nous pouvons déterminer les coordonnées X, Z grâce aux équations suivantes :

$$z = \frac{df_0}{p + f_0 \tan(\theta)}$$

$$x = z \tan(\theta)$$

où p est la position du rayon lumineux sur le détecteur, θ est l'angle de réflexion du rayon laser, d est la distance entre le miroir et la lentille, et f_0 est la distance focale de la lentille.

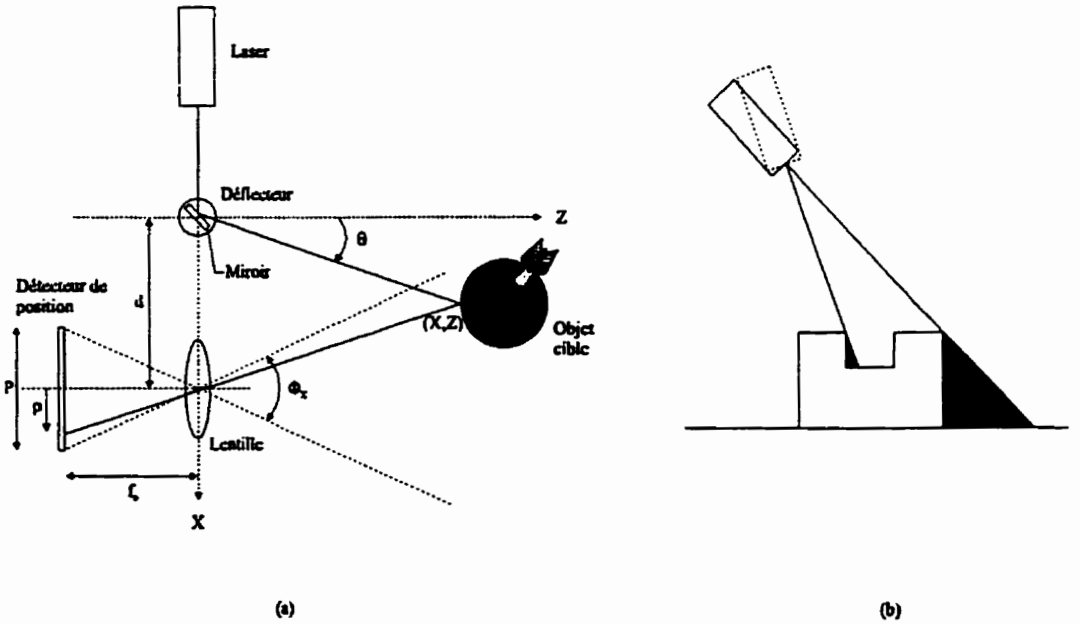


Figure 2.2 : (a) Principe de base de la triangulation active conventionnelle; (b) Effet d'ombrage.

Une des limitations de ce système est le compromis qu'il faut faire entre le champ de vision, la précision des mesures et l'effet d'ombrage (voir fig. 2.2b). En effet, les systèmes conventionnels basés sur la triangulation active utilisent une large ouverture angulaire, ce qui entraîne des problèmes d'ombrage. Une façon d'améliorer ce système optique est de synchroniser le faisceau avec le détecteur.

2.3 Balayage synchrone

Le balayage synchrone est une méthode pour obtenir un large champ de vision tout en ayant des angles de triangulation faibles. Cette réduction des angles de triangulation permet de diminuer l'effet d'ombrage, sans perdre de précision. Le principe est de synchroniser le faisceau lumineux du rayon laser avec le détecteur. Comme le montre la figure 2.3a, le champ de vision instantané du détecteur de position se déplace avec le champ de projection qui balaye la scène.

Rioux (1990) a introduit la triangulation auto-synchronisée basée sur le balayage synchrone (voir fig. 2.3b).

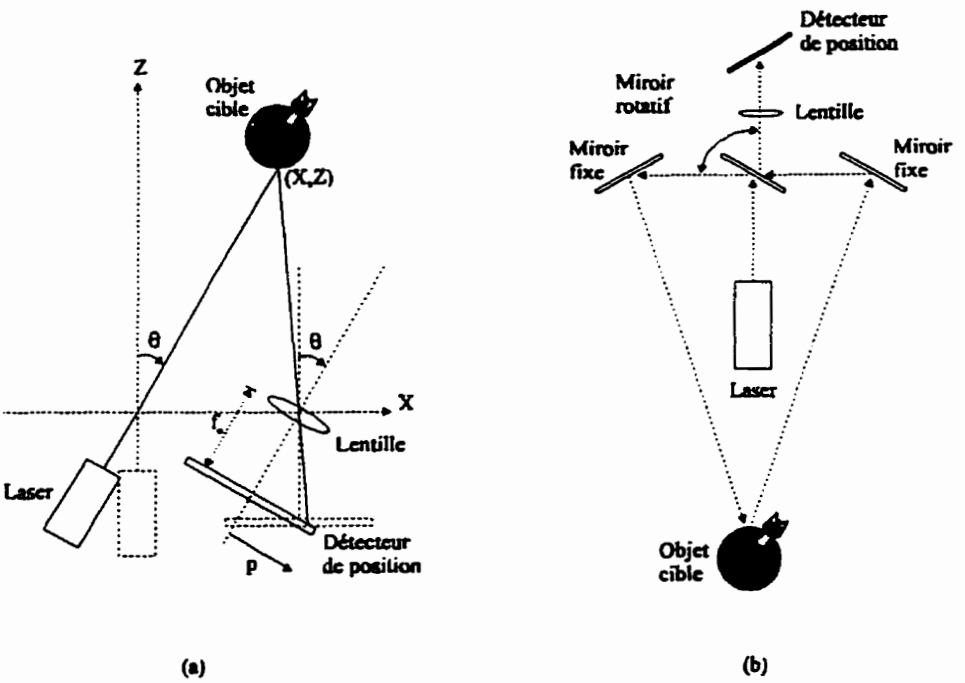


Figure 2.3 : Principe de base du balayage (a) synchronisé; (b) auto-synchronisé.

2.4 Auto-synchronisation

Le système de la figure 2.3b peut élégamment être étendu pour obtenir une image 3D. Il suffit d'ajouter un second axe de réflexion, perpendiculaire au premier, tel qu'illustré par la représentation schématique de la caméra auto-synchronisée de la figure 2.4. L'auto-synchronisation est obtenue à l'aide d'un miroir dont les deux faces sont réfléchissantes. La première face a pour fonction de réfléchir le rayon laser incident, tandis que la deuxième collecte la lumière reflétée par la scène. Cette approche permet de réduire la taille de la caméra et élimine totalement les problèmes de déphasage mécanique, ce qui permet d'augmenter la précision d'un facteur considérable. Les faibles angles de triangulation mis en jeu par la caméra auto-synchronisée, ont pour effet de diminuer considérablement le problème d'ombrage dont souffrait la technique de triangulation active conventionnelle. Le

processus d'acquisition d'une image produit alors trois quantités par échantillon : deux sont les positions angulaires des miroirs rotatifs (i,j) entier de 9 bits, et l'autre est la position du faisceau lumineux sur le détecteur de position (p) entier de 16 bits. Ce système est précis dans la mesure où les miroirs sont plats et où il n'y a aucune vibration sur les axes de rotation.

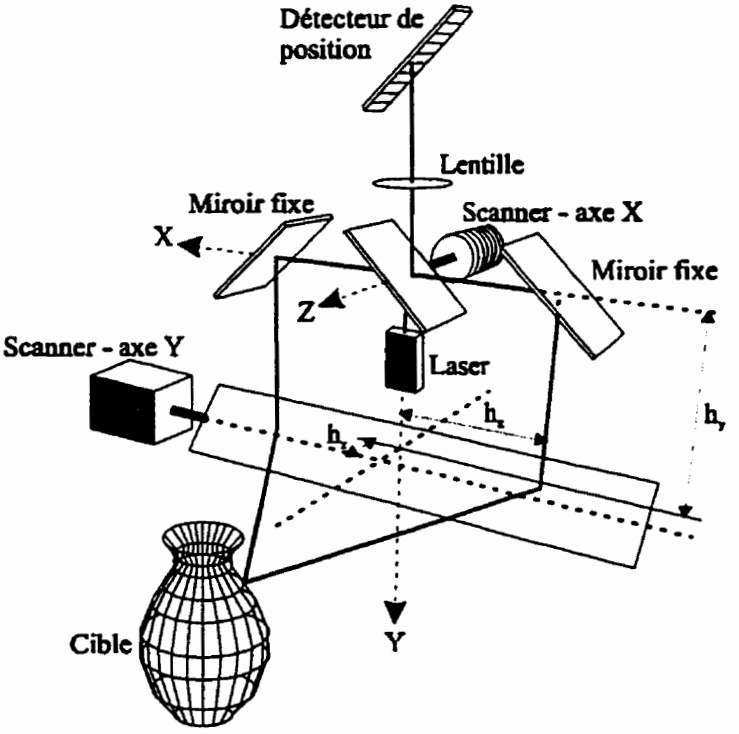


Figure 2.4 : Représentation schématique de la caméra auto-synchronisée

Les valeurs p, i et j sont les paramètres qui permettront d'obtenir une image de l'objet dans l'espace cartésien. Les équations qui permettent cette transformation, sont développées au chapitre suivant.

Chapitre 3 : Analyse des algorithmes de calcul

Dans ce chapitre, la complexité de deux algorithmes de calcul des coordonnées cartésiennes, associés à la caméra auto-synchronisée est comparée : le premier algorithme est appelé "algorithme Direct", parce qu'il découle directement de la géométrie du système optique, tandis que le deuxième, obtenu par approximation de l'algorithme Direct, est appelé "algorithme FIT". Cette comparaison permettra de choisir l'algorithme qui sera implanté sur les systèmes étudiés au chapitre 4. De plus, une méthode d'accélération basée sur l'utilisation de tables à valeurs pré-calculées sera étudiée afin de diminuer le temps de calcul de l'algorithme choisi.

3.1 Définition de l'environnement d'analyse

Le but de cette section est de formuler les hypothèses qui seront utilisées dans la suite de ce chapitre. Afin de ne pas biaiser les analyses qui seront faites dans les sections suivantes, il faut fixer le contexte de base pour les comparaisons.

Le but étant de comparer le temps d'exécution de deux algorithmes, il faut donner les spécifications d'un modèle théorique du processeur. Ce modèle aura néanmoins des caractéristiques comparables à celles de processeurs disponibles sur le marché. Ces spécifications formeront une base commune sur laquelle les comparaisons qui seront faites aux sections subséquentes de ce chapitre pourront se reposer.

Dans un premier temps, faisons l'hypothèse que l'utilisation de nombres à virgule flottante est nécessaire pour obtenir la précision voulue (voir section 3.7). Les processeurs modernes

possèdent souvent une unité arithmétique et un multiplieur indépendants afin de pouvoir effectuer une multiplication et une opération arithmétique en parallèle en un seul cycle d'horloge.

Le modèle du processeur, postulé dans cette section, possède une telle architecture qui fonctionne à 50 MHz et qui peut exécuter toutes les opérations arithmétiques élémentaires (multiplication, addition, etc.) en un cycle d'horloge. Les opérations exécutées en parallèle sur le multiplieur et sur l'unité arithmétique ne pourront avoir aucune variable commune, puisque les deux unités sont complètement indépendantes. L'algorithme de division, donné à l'annexe A (Cavanagh, 1984, p. 284), permet l'utilisation des opérations élémentaires rapides du modèle théorique. Cet algorithme calcule en fait l'inverse d'un nombre, en accédant à l'exposant ainsi qu'aux 8 bits les plus significatifs de la solution dans une table. Il converge ensuite vers le résultat à l'aide de l'algorithme Newton-Raphson qui utilise des opérations multiplication et addition (voir annexe A). Il faudra donc 8 cycles au modèle de processeur pour obtenir l'inverse d'un nombre avec une précision de 32 bits.

En résumé, le modèle théorique du processeur utilisé comme plate forme de référence pour évaluer la complexité des algorithmes, possède des unités opératives de 32 bits à virgule flottante, et il peut exécuter en parallèle une multiplication et une addition indépendantes l'une de l'autre, en un seul cycle d'horloge.

Avec cette définition du modèle de processeur, il est maintenant possible de calculer le nombre de cycles qu'il faut pour exécuter chacun des algorithmes, ce qui permet de comparer les résultats.

3.2 Algorithme DIRECT

La figure 2.4 du chapitre précédent montre le schéma de la caméra auto-synchronisée développée par les chercheurs du CNRC. Ce système optique ne fournit pas directement les coordonnées d'un point dans l'espace cartésien. En effet, les paramètres (p,i,j) que fournit la caméra auto-synchronisée sont respectivement, la position du faisceau lumineux reflété sur le détecteur, ainsi que les angles des deux miroirs rotatifs. Il est néanmoins possible d'obtenir la transformation de ces paramètres en des points (x,y,z) de l'espace cartésien. Béraldin (1993) a développé un ensemble d'équations, à partir de la géométrie de la caméra, équations qui permettent d'obtenir cette transformation. Ces équations supposent que les miroirs sont plats, sans défaut et qu'il n'y a aucune vibration sur les axes de rotation.

$$\begin{aligned} x(p,\theta,\phi) &= x_s(p,\theta) \\ y(p,\theta,\phi) &= [z_s(p,\theta) - h_x \cos(\gamma/2) - h_y] \sin(2\phi) - h_z \cos(2\phi) + h_y \\ z(p,\theta,\phi) &= [z_s(p,\theta) - h_x \cos(\gamma/2) - h_y] \cos(2\phi) - h_z \sin(2\phi) + h_z \end{aligned} \quad (1)$$

avec

$$\begin{aligned} x_s(p,\theta) &= X_{-\infty}(\theta) + P_{-\infty} \frac{X_0(\theta) - X_{-\infty}(\theta)}{P_{-\infty} - p} \\ z_s(p,\theta) &= Z_{-\infty}(\theta) + P_{-\infty} \frac{Z_0(\theta) - Z_{-\infty}(\theta)}{P_{-\infty} - p} \\ \phi(j) &= \phi_0 + \phi_1 j + \phi_3 j^3 \\ \theta(p,i) &= \theta_0 + \theta_1 i + \theta_2 i^3 + \theta_3 p^3 \end{aligned} \quad (2)$$

et

$$X_{-\infty}(\theta) = \frac{T\cos(\beta - 2\theta) + S\cos(\beta)\sin(\gamma/2 - \theta)}{\cos(\beta - \gamma)}$$
$$Z_{-\infty}(\theta) = \frac{S\cos(\beta)\cos(\gamma/2 - \theta) - T[\sin(\beta - 2\theta) + \sin(\beta - \gamma)]}{\cos(\beta - \gamma)} \quad (3)$$
$$X_o(\theta) = T\frac{\sin(2\theta)}{\sin(\gamma)}$$
$$Z_o(\theta) = -T\frac{\cos(2\theta) + \cos(\gamma)}{\sin(\gamma)}$$

où h_x, h_y, h_z, ϕ_0, ϕ_1, ϕ_3, θ_0, θ_1, θ_3, $P_{-\infty}$, β, γ, S et T sont des paramètres du système physique qui dépendent de sa géométrie.

L'angle entre le faisceau lumineux incident et celui reflété sur l'objet cible est dénoté γ. Cet angle est appelé "angle de triangulation" et il est considéré comme étant constant. Selon la calibration de la caméra (Béraldin, 1993), l'image peut ne pas atteindre toutes les cellules sur le détecteur de position. Le point maximal que peut atteindre le faisceau est appelé "point extrême" et il est dénoté $P_{-\infty}$. L'angle d'inclinaison du détecteur de position qui est déterminé selon la condition de Scheimplug (Béraldin, 1994) est représenté par β et est constant. $\phi(j)$ et $\theta(i)$ sont les angles associés aux mécanismes de balayage sur les axes x et y. Ces angles doivent être corrigés par des polynômes du 3ième ordre dû au caractère non-linéaire des mécanismes de balayage. Finalement, S est la distance entre la lentille et la position effective du pivot de l'axe de collection et T est la moitié de la distance entre le pivot de l'axe de collection et celui de l'axe de projection.

Khali (1995) discute le défi que représente la mise en oeuvre de ces équations. Pour mieux comprendre le problème, on peut borner la complexité de l'algorithme en termes de cycles nécessaires pour l'exécution de l'algorithme. Après simplification des équations (c'est-à-dire après avoir pré-calculé les constantes, factorisé, etc.) le système d'équations utilise 1 division,

86 multiplications et 69 additions. Ces nombres incluent le temps nécessaire pour calculer les fonctions trigonométriques requises (annexe A). En se basant sur le modèle de processeur décrit à la section précédente, il est possible d'effectuer 69 multiplications et additions en parallèle si on néglige les dépendances de données. Il restera donc 27 multiplications et 1 division à calculer. Il faudrait donc au moins 94 cycles au processeur virtuel pour terminer la transformation d'un point (p,i,j) dans l'espace cartésien. D'après ce calcul, il faudrait au moins trois processeurs basés sur notre modèle pour réussir à atteindre notre objectif de 1 million de points par seconde.

Cependant, comme il a été mentionné dans la description du modèle de processeur, pour atteindre la performance précédente, il faut que l'utilisation parallèle de l'unité arithmétique et du multiplieur se fasse de façon idéale, c'est-à-dire qu'il ne faut pas que les opérations utilisent de variables communes dans le même cycle. En effet, dû aux dépendances de données, il est improbable qu'un ordonnancement des opérations de cet algorithme puisse fournir, à chaque cycle, une multiplication et une addition indépendantes. Le nombre de cycles nécessaires pour exécuter la partie arithmétique de l'algorithme sera donc compris entre le nombre de cycles obtenu grâce à un ordonnancement parallèle, c'est-à-dire une multiplication et une addition à chaque cycle, et un ordonnancement séquentiel (une seule opération par cycle).

$$\underset{parallèle}{Ordonnancement} \leq Nb.\ de\ cycles \leq \underset{séquentiel}{Ordonnancement}$$

$$94 \leq Nb.\ de\ cycles \leq 163$$

Mentionnons ici que plusieurs algorithmes d'approximation de fonctions trigonométriques

peuvent être utilisés. Néanmoins, les performances qu'offre le modèle pour exécuter les opérations de multiplication et d'addition sur des nombres à virgule flottante, ont favorisé l'utilisation des séries de Taylor (voir annexe A). La précision qui peut être obtenue grâce aux séries de Taylor dépend du nombre de termes de la série qui sont calculés. Les angles que l'algorithme FIT devra calculer sont connus, puisqu'ils dépendent du paramètre *j* dont les valeurs sont discrètes. Pour chacun de ces angles, une première fonction décrite en langage C a été développée pour calculer la fonction trigonométrique réelle, et une seconde fonction en calcule la série de Taylor tronquée. Enfin, une troisième fonction calcule l'erreur obtenue par cette approximation. Ce programme a permis de déterminer que l'erreur obtenue lors de l'approximation des fonctions trigonométriques par les sept premiers termes de la série de Taylor était négligeable, puisque l'erreur maximale est de 1.4 dans un million. Après factorisation par la règle d'Hörner, les approximations de fonctions trigonométriques accaparent à elles seules 60 multiplications et 48 additions. De plus, il est difficile de les paralléliser, puisque chaque addition dépend de la multiplication précédente etc. Il est donc clair qu'un effort particulier doit être fait pour développer un algorithme qui demande le calcul d'un plus petit nombre de fonctions trigonométriques.

C'est pour palier à ce problème que les chercheurs du CNRC ont obtenu, par approximation de l'algorithme Direct, un ensemble d'équations beaucoup moins complexe à implanter, l'algorithme "FIT".

3.3 Algorithme FIT

Comme il a été vu dans la section précédente, l'algorithme Direct nécessite le calcul de nombreuses fonctions trigonométriques. Le but de cette section est de définir un algorithme qui a été obtenu par lissage de courbe et qui a pour but de réduire la complexité des transformations géométriques.

$$
\begin{aligned}
x &= x(p,i) = \frac{X_g(i)}{p - P_\infty} - X_0(i) \\
y &= y(p,i,\phi) = \left(\frac{Z_g(i)}{p - P_\infty} - Z_0(i) \right) \sin\phi + H_y(j) \\
z &= z(p,i,\phi) = \left(\frac{Z_g(i)}{p - P_\infty} - Z_0(i) \right) \cos\phi + H_z(j)
\end{aligned}
\tag{4}
$$

avec

$$
\begin{aligned}
X_g(i) &= \sum_{n=0}^{5} A_n i^n; \; X_0(i) = \sum_{n=0}^{5} B_n i^n \\
Z_g(i) &= \sum_{n=0}^{5} C_n i^n; \; Z_0(i) = \sum_{n=0}^{5} D_n i^n \\
H_y(j) &= \sum_{n=0}^{5} E_n j^n; \; H_z(j) = \sum_{n=0}^{5} F_n j^n
\end{aligned}
\tag{5}
$$

et

$$
\phi(j) = \phi_0 + \phi_1 j + \phi_3 j^3 \tag{6}
$$

Les paramètres (p, i, j) et $\phi(j)$ sont les mêmes que dans l'algorithme Direct. Notons que les séries peuvent être factorisées en utilisant la règle d'Hörner pour réduire le nombre d'opérations.

Par exemple :

$$X_g(i) = A_0 + A_1 i + A_2 i^2 + A_3 i^3 + A_4 i^4 + A_5 i^5$$
$$\textit{devient } X_g(i) = ((((A_5 i + A_4) i + A_3) i + A_2) i + A_1) i + A_0 \quad (7)$$

De même que pour l'algorithme Direct, l'algorithme FIT aura un nombre d'opérations compris entre celui d'un ordonnancement parallèle et celui d'un ordonnancement séquentiel. Selon les équations (4), (5) et (6), 52 multiplications, 49 additions et 1 division devront être calculées pour la transformation d'un point dans le plan cartésien. Dû encore une fois aux dépendances de données, il faudra au modèle de processeur défini à la section 3.1, 109 cycles dans le cas d'un algorithme séquentiel et 60 cycles dans le cas où toutes les additions sont faites en parallèle avec des multiplications.

$$60 \leq \textit{Nb. de cycles} \leq 109$$

Les deux algorithmes présentés dans les sections précédentes peuvent maintenant être comparés pour déterminer lequel peut être exécuté le plus rapidement.

3.4 Comparaison des algorithmes

Afin de déterminer l'algorithme qui devra être implanté, cette section compare les performances des deux algorithmes. Le modèle de processeur, défini à la section 3.1 de ce chapitre, sera encore une fois très utile pour comparer le nombre de cycles nécessaires à l'exécution des algorithmes. Le tableau 3.1 détaille le nombre et le type d'opérations pour chacun des algorithmes.

Tableau 3.1 : Nombre d'opérations pour chacun des algorithmes.

	Additions	Multiplications	Divisions	Opérations trigonométriques	nb. de cycles
Direct	21	26	1	8	94 @ 163
FIT	37	37	1	2	60 @ 109

Ces résultats ne tiennent compte que du temps de calcul qui dépend de la partie arithmétique de chacun des algorithmes. Cette comparaison ayant pour but de mettre en évidence l'avantage d'un algorithme par rapport à l'autre, seules les opérations élémentaires communes à la plupart des processeurs sont prises en compte (multiplication, addition, inverse).
Il est à noter que les divisions et les opérations trigonométriques ne sont pas des opérations élémentaires[1]. L'opération "division" est en fait 8 opérations élémentaires qui permettent d'obtenir une précision sur 32 bits (voir annexe A). La fonction *sinus* est calculée à l'aide d'une série qui demande 8 multiplications et 6 additions, tandis que la fonction *cosinus* nécessite 7 multiplications et 6 additions (voir annexe A).

Les résultats de la table 3.1 montrent que l'algorithme FIT est potentiellement capable de tranformer un triplet (p,i,j) en un point de l'espace cartésien 50% plus rapidement que l'algorithme Direct. Pour connaître le gain réel obtenu, les deux algorithmes ont été codés en langage C et simulés sur un processeur TMS320C40 de Texas Instruments, très populaire dans le domaine de la vision par ordinateur. Ce DSP ("Digital Signal Processor") à virgule flottante a été choisi pour sa capacité d'exécuter deux opérations en parallèle (multiplication-

[1] Les colonnes "Addition" et "Multiplication" de la table 3.1, ne tiennent pas compte des opérations élémentaires dûes aux divisions et aux fonctions trigonométriques.

addition). Pour faire cette comparaison, les deux algorithmes ont été compilés avec le même compilateur et les mêmes options d'optimisation. De plus, les mêmes fonctions d'approximation pour calculer les fonctions trigonométriques ont été utilisées. La fréquence de l'horloge externe du processeur est de 50 MHz. Néanmoins, les unités opératives du TMS320C40 se réfèrent à l'horloge "machine" qui est définie par Texas Instruments (Texas Instruments, 1991) comme étant la moitié de la fréquence de l'horloge externe (25 MHz). Le nombre de cycles fourni par le simulateur a donc été multiplié par deux pour connaître la performance du DSP par rapport à l'horloge externe.

Tableau 3.2 : Résultats pratiques sur DSP.

	Direct	**FIT**
C40	840 cycles d'horloge 59,523 points/sec	416 cycles d'horloge 120,192 points/sec

Il est donc possible de conclure que l'algorithme FIT est environ 2 fois plus rapide que l'algorithme Direct, ce qui confirme l'analyse théorique. Il est maintenant possible de conclure que l'algorithme qui devra être implanté, est l'algorithme FIT. Dans le reste de ce travail, seul cet algorithme sera discuté.

La section suivante propose une méthode pour accélérer le temps d'exécution de l'algorithme et pour limiter l'accumulation des erreurs dues aux approximations successives.

3.5 Accélération par des tables de références

Le principe d'accélération par des tables LUTs ("Look-Up Tables"), consiste à pré-calculer toutes les valeurs que peut prendre un facteur, afin de les mettre dans un tableau. Par

exemple, si n est un entier de 16 bits, on peut calculer toutes les valeurs que peut prendre une fonction $R(n)$ puisque n est borné entre 0 et 65,535. Les résultats de ces calculs peuvent être placés dans une table en mémoire. Si n est une entrée du système, on peut accéder au résultat de $R(n)$ en utilisant directement n comme index qu'on additionne à l'adresse de base de la table dans la mémoire.

Le gain obtenu grâce à cette méthode est d'autant plus grand que le nombre d'opérations pour calculer $R(n)$ est élevé. En effet, peu importe la complexité de la fonction, on peut obtenir le résultat dans un temps égal au temps d'un accès mémoire.

Un autre avantage de cette méthode, c'est l'augmentation de la précision des résultats. Si, par exemple, $R(n)$ découle du développement en série de Taylor qui fait l'approximation de la fonction *sinus*, la précision du résultat dépendra du nombre de termes de la série qui seront calculés. Si on doit calculer cette approximation pendant l'exécution de l'algorithme, pour obtenir un résultat dans un temps raisonnable, il faudra se restreindre aux sept ou huit premiers termes. Mais si la méthode d'accélération par LUTs est utilisée, la fonction pourra être calculée avec un plus grand nombre de termes puisque le calcul ne se fera pas en temps réel. De plus, cette méthode exige moins d'opérations et peut produire un résultat final aussi précis avec moins de chiffres significatifs, car il n'y a pas de perte de précision cumulative dans les calculs.

Le désavantage des LUTs est l'espace mémoire nécessaire pour garder toutes ces valeurs pré-calculées. Pour reprendre l'exemple de la fonction $R(n)$, si les valeurs sont représentées sur 32 bits, il faudra (2^{16} valeurs * 4 octets/valeur) = 262,144 octets de mémoire, ce qui est abordable. Par contre, si une fonction dépend de deux paramètres n et m représentés sur 16

bits, et qu'on cherche à l'accélérer par une LUT, la fonction demandera $(2^{16} * 2^{16} * 4) = 17$ Goctets de mémoire ce qui n'est pas raisonnable. Il faut donc vérifier que l'accélération d'un algorithme par cette méthode n'a pas un impact démesuré sur la quantité de mémoire requise.

3.6 Accélération de l'algorithme FIT

Comme nous venons de le voir à la section précédente, s'il fallait mettre $X(p,i)$, $Y(p,i,j)$, et $Z(p,i,j)$ dans des LUTs, pour qu'il n'y ait aucun calcul à faire, il faudrait plus de deux millions de giga-octets. On peut, néanmoins, utiliser cette méthode pour accélérer certaines sous-fonctions et ainsi diminuer le nombre de calculs à faire lors de l'exécution. Il serait possible de mettre les deux fonctions trigonométriques de l'algorithme FIT dans des LUTs. Nous pourrions ainsi accéder à ces fonctions directement par la variable d'entrée j, qui correspond à l'angle du miroir rotatif sur l'axe y. Il serait aussi possible de mettre la division sous forme de table, puisque la division ne dépend que de la variable d'entrée p, qui est la position du faisceau lumineux sur le détecteur. Et finalement, les six développements en série de l'algorithme peuvent être mis en LUTs, puisque chacune des séries ne dépend que d'une seule variable. Il y aurait ainsi autant de tableaux qu'il y a de facteurs pré-calculés, et chacun de ces tableaux aurait autant de lignes que le vecteur d'entrée permet d'adresser. Il serait donc possible d'utiliser directement p, i ou j comme étant des indices qu'on additionnerait à l'adresse de base d'une table. Par exemple, dans le cas où i est un nombre de 16 bits, il serait possible d'accéder à 65,536 valeurs pré-calculées de chacun des facteurs qui dépendent de i.

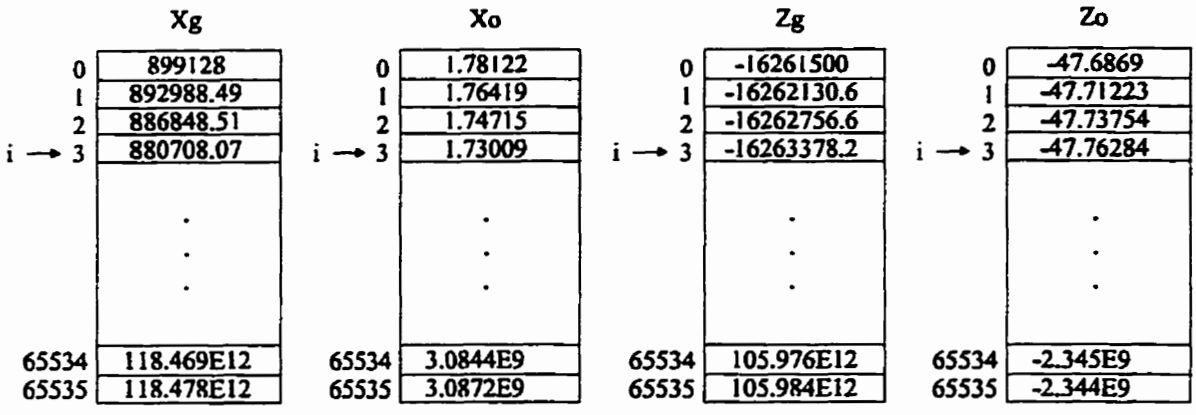


Figure 3.1 : Exemple de tables de référence

Pour chacune de ces séries, seulement le temps d'un accès mémoire est nécessaire pour obtenir le résultat, au lieu de 10 cycles lorsqu'il faut faire le calcul. Il en va de même pour les deux fonctions trigonométriques, qui pourront être accédées directement par j, sans avoir à calculer ϕ, et qui demandaient chacune 13 ou 14 cycles. Finalement, il est possible de remplacer la fonction qui calcule l'inverse de $(p - P_{-})$, et qui demandait 8 cycles (voir annexe A), par un accès mémoire.

Dans le cas où i, j et p sont sur 16 bits et que les facteurs qui dépendent d'une seule variable sont mis dans des LUTs, il faudra 9 tables de 65,536 nombres chacune. Chacun de ces nombres étant représentés sur 32 bits, il faudra 4.7 Moctets de mémoire. En faisant l'hypothèse que le budget alloué pour le système permette de mettre cette quantité de mémoire statique (SRAM), un accès mémoire sera fait en un seul cycle. Pour transformer un triplet (p,i,j) en un point du plan cartésien avec l'utilisation de LUTs, il faudrait au moins 9 cycles dans le cas d'un ordonnancement parallèle, et 18 cycles dans le cas d'un

ordonnancement séquentiel. Le gain obtenu serait donc plus de 6 fois plus rapide que l'algorithme FIT sans accélération.

Pour vérifier ces résultats théoriques, l'algorithme FIT, utilisant des LUTs, a été décrit en langage C. Il est possible de comparer les résultats obtenus grâce à ce programme et celui qui a été obtenu dans la section 3.4 de ce chapitre.

Tableau 3.3 : Résultats pratiques sur DSP de l'accélération par LUTs.

	FIT (voir section 3.4)	FIT avec LUTs
C40	416 cycles 120,192 points/sec	164 cycles 304,878 points/sec

L'algorithme FIT est donc 2.5 fois plus rapide lorsqu'il est accéléré par l'utilisation de LUTs, mais au dépend d'une augmentation de l'espace mémoire requis.

3.7 Détermination du format des nombres

Un des buts de ce travail, est d'examiner plusieurs architectures de systèmes qui pourront atteindre l'objectif initial de transformer 1 million de triplets (p,i,j) par seconde en des points de l'espace cartésien. Un autre objectif qui découle du précédent, est de déterminer lequel de ces systèmes aura le coût le plus faible. Jusqu'à maintenant, il n'a été question que de nombres à virgule flottante représentés sur 32 bits. L'avantage de cette représentation est bien sûr la grande précision des résultats, mais il y a un coût à payer. En effet, un multiplieur à virgule flottante prend beaucoup plus d'espace sur un circuit qu'un multiplieur à virgule fixe dont les vecteurs d'entrées ont le même nombre de bits.

Dans le cas de la conception d'un circuit dédié (ASIC), il serait possible d'implanter plusieurs

unités opératives à virgule fixe, là où une seule unité à virgule flottante pourrait l'être. L'algorithme utilisé pour le calcul des coordonnées cartésiennes n'implique aucune relation directe ou indirecte avec un calcul précédent ou suivant celui-ci. Il est ainsi possible de faire plusieurs transformations sur des points indépendants en parallèle, et donc d'augmenter le débit total du système.

Format à virgule fixe

Peu importe que le nombre représenté soit très grand ou très petit, la répartition des bits entre la partie entière et la partie fractionnaire d'une représentation en virgule fixe ne varie pas. Il faut donc utiliser une mise à l'échelle qui n'est pas exprimée explicitement avec le nombre afin d'éviter les débordements, contrairement à une représentation en virgule flottante, où la mise à l'échelle est exprimée par l'exposant. La mise à l'échelle a deux fonctions : l'une est de s'assurer que la partie entière d'un nombre puisse être représentée sans débordement par le nombre de bits alloué à la mantisse, et l'autre est de minimiser la perte de précision due à la représentation d'un nombre dont la partie fractionnaire ne pourrait être représentée par le nombre de bits alloué à cet effet.

Par exemple, une représentation à virgule fixe qui possède trois bits de partie entière et un bit de partie fractionnaire ne peut pas représenter directement le nombre 9 (1001.0_2) puisque la partie entière exige quatre bits. Il est néanmoins possible de représenter 9 par 4.5 (100.1_2) avec une mise à l'échelle de 2, qui devra être prise en compte lors des calculs subséquents. La représentation du nombre 0.75 (0.11_2) à l'aide de ce format implique une perte de précision, puisqu'il n'y a qu'un seul bit réservé pour la partie fractionnaire. Cependant,

encore une fois, il est possible de représenter 1.5 (001.1_2) avec une mise à l'échelle de 2^{-1}, ce qui correspond à un décalage vers la gauche de la valeur réelle.

Il est évident que les deux fonctions de mise à l'échelle sont mutuellement exclusives, puisque l'une décale les bits vers la droite et l'autre vers la gauche. Néanmoins, le décalage vers la droite a priorité, puisque la perte du bit le plus significatif de la partie entière a une conséquence bien plus grande que la perte du bit le moins significatif de la partie fractionnaire. Cette priorité implique une perte de précision potentielle associée à l'utilisation d'un format à virgule fixe et l'impact de cette perte de précision devra être calculé pour en déterminer l'importance sur le résultat final.

Détermination de l'erreur inhérente à l'utilisation d'un format à virgule fixe

Pour déterminer cette erreur, il faut identifier les endroits de l'algorithme où les pertes de précision surviennent. Il faut examiner chaque étape du processus d'exécution des opérations.

La multiplication de deux nombres à virgule fixe consiste simplement à multiplier les mantisses des deux nombres et à additionner leur mise à l'échelle respective. La taille de la mantisse du résultat d'une multiplication est déterminée par l'addition de la taille des deux nombres à multiplier. Par exemple, le résultat de la multiplication de deux nombres à virgule fixe représentés sur 24 bits aura une taille de 48 bits. Pour qu'aucune perte de précision ne se produise, il faudrait garder tous ces bits. Il est clair qu'en pratique ce n'est pas possible, puisqu'à chaque multiplication subséquente le nombre de bits augmenterait de 24 et le résultat final aurait une taille inacceptable pour une implantation physique du système. Il faut donc,

après chaque multiplication, décaler le résultat pour qu'il soit remis sur 24 bits, et rajuster la mise à l'échelle pour qu'elle tienne compte de ce décalage. Ce processus contribue à la perte de précision finale de l'algorithme, puisqu'à chaque fois, le décalage élimine un certain nombre de bits.

Dans le cas d'une addition, l'échelle des deux opérandes doit être égale, ce qui implique le décalage d'un ou des deux opérandes. Une fois ce processus terminé, il suffit d'additionner les mantisses en prenant garde qu'il n'y ait pas de débordement.

Pour déterminer la perte de précision due à l'utilisation d'un format à virgule fixe, une étude approfondie de l'algorithme FIT doit être faite pour obtenir les facteurs de mise à l'échelle optimaux à chaque étape du calcul. Cette étude doit faire les compromis idéaux entre le débordement lorsque le résultat est maximum, et la perte de précision lorsque le résultat est minimum.

Pour montrer les étapes nécessaires pour faire cette étude, considérons les facteurs A_n de l'équation (7) donnés au tableau 3.4. Ces facteurs sont ceux obtenus lors de la calibration de la caméra pour un objet de 1 mètre cube à une distance de 1 mètre. Pour simplifier la lecture, l'équation (7) est retranscrite ici.

$$X_g(i) = ((((A_5 i + A_4)i + A_3)i + A_2)i + A_1)i + A_0$$

Tableau 3.4 : Facteurs A_n, d'un objet de 1m³ à une distance d'un mètre, utilisés pour le calcul de X_g

	Sans mise à l'échelle	Avec mise à l'échelle
A_0	889,128	$28{,}097.75 \times 2^{5}$
A_1	-6,139.27	$-196{,}456.625 \times 2^{-5}$
A_2	-0.235729	$-247{,}179.75 \times 2^{-20}$
A_3	8.94595×10^{-5}	$96{,}056.40 \times 2^{-30}$
A_4	-4.64256×10^{-7}	$-255{,}227.406 \times 2^{-39}$
A_5	1.05075×10^{-10}	$236{,}607.84375 \times 2^{-51}$

Si une représentation en virgule fixe ayant 1 bit de signe, 18 bits de partie entière et 5 bits de partie fractionnaire est utilisée pour calculer X_g, le calcul aurait la forme suivante dans le cas où i est représenté sur 9 bits (la borne inférieure est calculée avec $i = 1$, puisque le calcul avec $i = 0$ est trivial) :

$$X_{gl_{i=511}} = A_5 * 511 = 236{,}607.84375 \times 2^{-51} * 511 = 120{,}906{,}608.1563 \times 2^{-51}$$
$$X_{gl_{i=1}} = A_5 * 1 = 236{,}607.84375 \times 2^{-51} * 1 = 236{,}607.84375 \times 2^{-51}$$

La valeur maximale que peut représenter la partie entière sur 18 bits, est $2^{18} = 262{,}144$. Il faut donc décaler les résultats puisque la borne supérieure ne peut pas être représentée par ce format. Pour une perte de précision minimale, il faut faire le minimum de décalage requis pour que le nombre puisse être représenté.

$$X_{gl_{i=511}} = 120{,}906{,}608.1563 \times 2^{-51} \xrightarrow[\text{9 décalages vers la droite}]{} 236{,}145.7190 \times 2^{-42}$$

$$X_{gl_{i=1}} = 236{,}607.84375 \times 2^{-51} \xrightarrow[\text{9 décalages vers la droite}]{} 462.12469 \times 2^{-42}$$

Il faut ensuite additionner à ces résultats la constante suivante, A_4, qui doit avoir le même

facteur d'échelle. Si A_4 doit être représentée avec une mise à l'échelle de 2^{-42}, la partie entière ne peut pas être représentée sur 18 bits. La mise à l'échelle minimale que peut prendre A_4 est de 2^{-39}. Il faut donc redécaler X_{g1} de 3 bits vers la droite.

$$X_{g2_{i=511}} = X_{g1} + A_4 = 29{,}518.214 \times 2^{-39} + (-255{,}227.406 \times 2^{-39}) = -225{,}709.19 \times 2^{-39}$$
$$X_{g2_{i=1}} = X_{g1} + A_4 = 57.765 \times 2^{-39} + (-255{,}227.406 \times 2^{-39}) = -255{,}169.64 \times 2^{-39}$$

Il faut bien sûr s'assurer que le résultat de l'addition puisse être représenté par le format défini plus haut. Il faut maintenant multiplier X_{g2} par i.

$$X_{g3_{i=511}} = X_{g2} * 511 = -225{,}709.19 \times 2^{-39} * 511 = -115{,}337{,}396.79 \times 2^{-39}$$
$$X_{g3_{i=1}} = X_{g2} * 1 = -255{,}169.64 \times 2^{-39} * 1 = -255{,}169.64 \times 2^{-39}$$

La partie entière étant représentée sur 18 bits, le résultat de la borne supérieure ne peut pas être représenté par ce format. Il faut donc effectuer 9 décalages vers la droite pour garder le maximum de précision possible.

$$X_{g3_{i=511}} = -115{,}337{,}396.79 \times 2^{-39} \xrightarrow[\text{9 décalages vers la droite}]{} -225{,}268.353 \times 2^{-30}$$

$$X_{g3_{i=1}} = -255{,}169.64 \times 2^{-39} \xrightarrow[\text{9 décalages vers la droite}]{} -498.378 \times 2^{-30}$$

La constante A_3, qui doit être additionnée à X_{g3}, peut être représentée sur 24 bits avec une mise à l'échelle de 2^{-30}, il n'y aura donc pas de décalage à faire avant l'addition.

$$X_{g4_{i=511}} = X_{g3} + A_3 = -225{,}268.35 \times 2^{-30} + 96{,}056.40 \times 2^{-30} = -129{,}211.94 \times 2^{-30}$$
$$X_{g4_{i=1}} = X_{g3} + A_3 = -498.378 \times 2^{-30} + 96{,}056.40 \times 2^{-30} = 95{,}558.028 \times 2^{-30}$$

Ces résultats peuvent être représentés avec le format à virgule fixe, aucun décalage de X_{g3} n'est donc nécessaire. Il faut maintenant multiplier X_{g4} par i.

$$X_{g5_{i-511}} = X_{g4} * 511 = -129{,}211.947 \times 2^{-30} * 511 = -66{,}027{,}304.842 \times 2^{-30}$$
$$X_{g5_{i-1}} = X_{g4} * 1 = 95{,}558.028 \times 2^{-30} * 1 = 95{,}558.028 \times 2^{-30}$$

Le résultat de la borne supérieure doit être décalé pour que la partie entière soit représentée par 18 bits. La constante, qui devra être additionnée au résultat de ce décalage, peut être représentée avec une mise à l'échelle minimale de 2^{-20}. Par conséquent, il faudra décaler X_{g5} de 10 bits vers la droite pour ne pas perdre trop de précision.

$$X_{g5_{i-511}} = -66{,}027{,}304.84 \times 2^{-30} \underset{\text{10 décalages vers la droite}}{\Longrightarrow} -64{,}479.79 \times 2^{-20}$$

$$X_{g5_{i-1}} = 95{,}558.028 \times 2^{-30} \underset{\text{10 décalages vers la droite}}{\Longrightarrow} 93.318 \times 2^{-20}$$

Il est maintenant possible d'additionner X_{g5} avec A_2 puisque les deux ont la même mise à l'échelle.

$$X_{g6_{i-511}} = X_{g5} + A_2 = -64{,}479.79 \times 2^{-20} + (-247{,}179.75) \times 2^{-20} = -311{,}659.54 \times 2^{-20}$$
$$X_{g6_{i-1}} = X_{g5} + A_2 = 93.318 \times 2^{-20} + (-247{,}179.75) \times 2^{-20} = -247{,}086.43 \times 2^{-20}$$

Après l'addition, la partie entière du résultat de la borne supérieure ne peut pas être représentée par 18 bits. Ainsi, il faut redécaler X_{g5} et A_2 de 1 bit pour corriger ce problème.

$$X_{g5_{i-511}} = -66{,}027{,}304.84 \times 2^{-30} \underset{\text{11 décalages vers la droite}}{\Longrightarrow} -123{,}589.875 \times 2^{-19}$$

$$X_{g5_{i-1}} = 95{,}558.028 \times 2^{-30} \underset{\text{11 décalages vers la droite}}{\Longrightarrow} 46.659 \times 2^{-19}$$

Une fois qu'on s'est assuré que la partie entière du résultat pourra être représentée sur 18 bits l'addition peut maintenant être faite.

$$X_{g6_{i=511}} = X_{g5} + A_2 = -32{,}239.89 \times 2^{-19} + (-123{,}589.875) \times 2^{-19} = -155{,}829.77 \times 2^{-19}$$
$$X_{g6_{i=1}} = X_{g5} + A_2 = 46.659 \times 2^{-19} + (-123{,}589.875) \times 2^{-19} = -123{,}543.215 \times 2^{-19}$$

Ces résultats doivent maintenant être multipliés par i.

$$X_{g7_{i=511}} = X_{g6} * 511 = -155{,}829.77 \times 2^{-19} * 511 = -79{,}629{,}012.44 \times 2^{-19}$$
$$X_{g7_{i=1}} = X_{g6} * 1 = -123{,}543.215 \times 2^{-19} * 1 = -123{,}543.215 \times 2^{-19}$$

Il faut à nouveau tenir compte de la constante A_1 pour déterminer le nombre de décalage qu'il faut faire sur X_{g7}, afin qu'elle puisse être représentée par le format des nombres défini plus tôt.

$$X_{g7_{i=511}} = -79{,}629{,}012.44 \times 2^{-19} \xrightarrow[\text{14 décalages vers la droite}]{} -4{,}860.169 \times 2^{-5}$$

$$X_{g7_{i=1}} = -123{,}543.215 \times 2^{-19} \xrightarrow[\text{14 décalages vers la droite}]{} -7.540 \times 2^{-5}$$

Il est maintenant possible de faire l'addition de X_{g7} avec A_1.

$$X_{g8_{i=511}} = X_{g7} + A_1 = -4{,}860.169 \times 2^{-5} + (-196{,}456.625) \times 2^{-5} = -201{,}316.79 \times 2^{-5}$$
$$X_{g8_{i=1}} = X_{g7} + A_1 = -7.540 \times 2^{-5} + (-196{,}456.625) \times 2^{-5} = -196{,}464.165 \times 2^{-5}$$

Ces résultats pouvant être représentés sur 18 bits de partie entière, il faut les multiplier par i.

$$X_{g9_{i=511}} = X_{g8} * 511 = -201{,}316.79 \times 2^{-5} * 511 = -102{,}872{,}881.836 \times 2^{-5}$$
$$X_{g9_{i=1}} = X_{g8} * 1 = -196{,}464.165 \times 2^{-5} * 1 = -196{,}464.165 \times 2^{-5}$$

Le résultat de la borne supérieure implique qu'il faut un décalage de 10 bits vers la droite pour garder le plus de bits possibles. De plus, la partie entière de A_0 peut être représentée par 18 bits.

$$X_{g9_{i=511}} = -102{,}872{,}881.836 \times 2^{-5} \xrightarrow[10\ \textit{décalages vers la droite}]{} -100{,}461.80 \times 2^{5}$$

$$X_{g9_{i=1}} = -196{,}464.165 \times 2^{-5} \xrightarrow[10\ \textit{décalages vers la droite}]{} -191.860 \times 2^{5}$$

Finalement, ces résultats doivent être additionnés à A_0 pour obtenir le résultat final.

$$X_{g_{i=511}} = X_{g9} + A_0 = -100{,}461.80 \times 2^{5} + 28{,}097.75 \times 2^{5} = -72{,}364.048 \times 2^{5}$$
$$X_{g_{i=1}} = X_{g9} + A_0 = -191.86 \times 2^{5} + 28{,}097.75 \times 2^{5} = 27{,}905.890 \times 2^{5}$$

Afin de ne pas dégrader la précision, la valeur finale de X_g ne sera pas calculée. La dernière mise à l'échelle devra donc être prise en compte dans la suite des calculs. L'équation finale que nous obtenons, est :

$$X_g(i) = (((((A_5 i \times 2^{-12} + A_4) i \times 2^{-9} + A_3) i \times 2^{-11} + A_2) i \times 2^{-14} + A_1) i \times 2^{-10} + A_0) \times 2^{5}$$

Les décalages représentés par 2^x, ont ainsi deux fonctions : la première est de remettre le résultat de la multiplication sur 24 bits, et la seconde est de s'assurer que l'addition sera effectuée avec deux nombres ayant le même facteur de mise à l'échelle.

De la même façon que le développement de cette équation a été obtenu, il est possible d'obtenir les équations reliées au reste du calcul. Un résumé des étapes de ces développements se trouve en annexe D. De plus, un programme a été décrit en VHDL pour obtenir la perte de précision exacte qui est obtenue avec cet algorithme (voir annexe E).

Ce programme a permis de déterminer qu'une représentation des nombres sur 24 bits, à virgule fixe, donne une perte de précision de 5 dans 100,000 en erreur absolue.

Représentation virgule fixe avec LUTs

Béraldin (1993) montre que la calibration de la caméra est une fonction du volume de l'objet cible, et de sa distance par rapport à la caméra. En fait, si la caméra est calibrée pour un volume de 1 mètre cube, il faudra recalibrer la caméra pour visualiser un objet dont le volume est différent. Cette recalibration entraîne une modification des facteurs ($A_n B_n C_n D_n E_n F_n$) dont dépendent les séries de l'algorithme FIT. Les décalages déterminés par rapport à ces constantes étant fixes, si la calibration de la caméra est modifiée, ou si une quelconque modification sur la taille des opérandes est faite, l'étude qui permet de déterminer l'échelle des facteurs devra être refaite, et le circuit dédié conçu basé sur cet algorithme devra être modifié.

L'utilisation de LUT pour accélérer le calcul permet de remédier de façon inhérente, au problème de recalibration. En effet, puisque les résultats du développement en série qui dépendent de la calibration de la caméra sont dans une mémoire, la seule modification qu'entraînera la recalibration sera de recharger la mémoire avec de nouvelles valeurs.

La perte de précision de l'algorithme provient surtout des décalages effectués après chaque multiplication. L'utilisation des LUTs élimine les calculs des séries, ce qui diminue le nombre de décalages, et implique une augmentation de la précision des résultats. Ainsi, pour obtenir la même perte de précision que l'algorithme sans accélération par table, seulement 20 bits à virgule fixe seront nécessaires pour représenter les nombres.

L'algorithme sans accélération par LUT nécessite l'utilisation d'un décaleur pour permettre de fournir, à l'additionneur, des nombres qui ont une mise à l'échelle égale. Ce décaleur doit

être variable puisque, comme il est montré dans l'analyse précédente, le nombre de décalages n'est pas toujours le même après chacunes des multiplications.

Une étude similaire à celle de la section précédente a permis de déterminer que l'utilisation de LUT nécessite un décalage fixe de 17 bits, après chaque multiplication de l'algorithme. Cette propriété a des conséquences intéressantes, puisqu'elle permet de faire les décalages de façon câblée, et ainsi d'éliminer les décaleurs.

Le prochain chapitre étudiera plusieurs architectures, dont une qui utilise un format à virgule fixe avec LUT, ce qui permettra l'implantation de plusieurs unités opératives en parallèle, vu la taille réduite des unités opératives à virgule fixe par rapport à une unité opérative à virgule flottante.

Chapitre 4 : Architectures de systèmes

La caméra auto-synchronisée fournit au système de calcul les deux angles (i,j) des miroirs rotatifs, ainsi que la position sur le détecteur (p) du faisceau lumineux. Ce triplet (p,i,j) doit ensuite être transformé en un point (x,y,z) de l'espace cartésien. Les algorithmes vus au chapitre précédent font cette transformation indépendamment des points qui précèdent et qui suivent le point traité. Cette propriété est intéressante puisqu'elle permet la transformation de plusieurs triplets (p,i,j) de façon parallèle, et ainsi d'augmenter le débit de traitement.

Le chapitre précédent a permis de déterminer que de meilleures performances pourront être obtenues par l'implantation de l'algorithme FIT plutôt que par l'implantation de l'algorithme Direct. De plus, il a été démontré que l'utilisation de LUTs permet d'accélérer le calcul de façon très intéressante.

Ce chapitre a donc pour but d'étudier et de développer plusieurs implantations de l'algorithme FIT. Le premier système étudié est basé sur une architecture mixte matérielle-logicielle. Cette implantation prend comme base une carte possédant trois processeurs, un circuit dédié et une SRAM ("Static Random Access Memory") dans laquelle seront gardés en mémoire les LUTs.

Les deuxième et troisième systèmes étudiés dans ce chapitre sont des versions logicielles de l'algorithme. L'une est basée sur un processeur de Texas Instruments, le TMS320C40 et l'autre sur un processeur de Analog Devices, le ADSP-21060.

Finalement, une version matérielle sera étudiée. Ce système sera développé sur un FPGA ("Field Programmable Gate Array") de la famille Xilinx. Cette architecture utilisera une

représentation des nombres en virgule fixe vu la complexité d'implantation des unités opératives à virgule flottante.

4.1 Implantation mixte matérielle - logicielle

Au début de ce projet, l'algorithme FIT n'était pas tout à fait le même que celui présenté au chapitre 3. La différence entre les deux provient de la correction qui était faite sur le point obtenu du détecteur de position. Cette correction a été par la suite éliminée, puisque l'erreur commise en utilisant l'algorithme FIT au lieu de l'algorithme Direct était plus grande que l'influence de la correction sur p. Cette correction est montrée à l'équation suivante:

$$p'(p,i) = p + k_1\, i + k_2\, i^3 + k_3\, i^5 \quad avec\ k_n = constantes$$

La mise en oeuvre de cette équation ne représente en elle même que quelques cycles supplémentaires. Néanmoins, son impact sur l'algorithme est important.

La division $1/(p - P_w)$ de l'équation (4) ne peut donc plus être mise dans une LUT, puisque p y est remplacé par $p'(p,i)$. Comme il a été montré à la section 3.5 du chapitre précédent, une fonction qui dépend de plus d'une variable peut difficilement être mise dans une LUT vu la taille que devra avoir la mémoire. De plus, il n'est pas possible d'utiliser p' comme index pour accéder à une LUT puisque ce n'est pas un nombre entier, mais un nombre à virgule flottante sur 32 bits. Il faut donc calculer $p'(p,i)$ en plus d'effectuer la division, ce qui augmente considérablement la complexité de calcul de l'algorithme FIT. Cette correction sur p sera donc prise en compte lors de cette implantation, mais pas dans les autres.

À la section 3.7 de ce mémoire, il est montré que si une application demande une très grande

précision, l'implantation devra être faite avec une représentation des nombres en virgule flottante. Ainsi, le besoin d'un diviseur à virgule flottante pousse vers une implantation logicielle de l'algorithme, vu la grande complexité et les mauvaises performances d'une telle unité.

L'utilisation d'une carte possédant plusieurs processeurs permet de casser le goulot d'étranglement, qui dans ce cas est le grand nombre d'opérations à effectuer ainsi que les accès aux LUTs. Pour accélérer les accès aux LUTs, un circuit dédié peut agir comme un générateur d'adresses qui distribue les données aux différents processeurs.

Un système basé en partie sur la carte Blazer (Image & Signal Processing, 1994) semble donc être une solution rapide à développer, et qui offre de bonnes performances. Cette carte dispose de trois DSPs ("Digital Signal Processing") TMS320C40 de Texas Instruments en parallèle, interconnectés entre eux par des ports de communication. Chaque processeur possède en plus un port de communication externe qui peut être branché à un circuit dédié à la distribution des données. L'architecture de ce système est montrée à la figure 4.1.

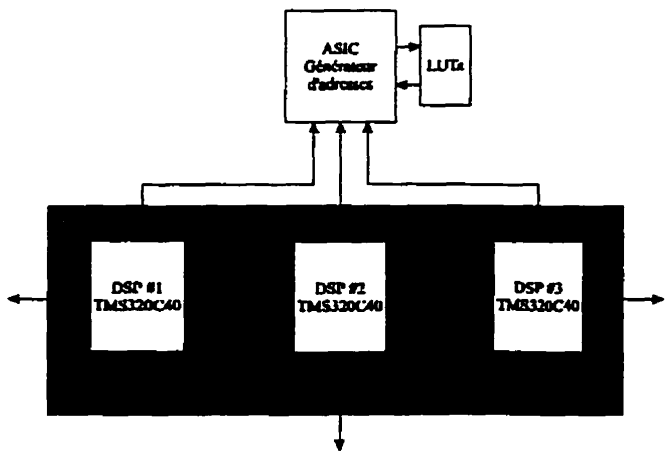


Figure 4.1 : Architecture d'un système basé sur la carte Blazer.

Pour implanter ce système, le circuit dédié doit disposer de trois ports de communication externes compatibles à ceux des processeurs. Le port de communication qu'utilise le TMS320C40 est un port de 8 bits asynchrone bidirectionnel. L'implantation de ce port étant très complexe, il a été modifié afin d'être synchrone avec l'horloge interne du circuit dédié. L'effet de cette synchronisation n'aura qu'un très faible impact sur la vitesse d'une communication. En fait, dans le pire cas, une communication aura besoin d'un cycle supplémentaire. Par contre, la faible complexité du circuit permet d'avoir une fréquence d'horloge deux fois plus élevée que celle requise pour une communication asynchrone. La fréquence plus élevée permet de diminuer le délai additionnel dû à l'attente d'un front sur l'horloge du circuit dédié.

Une architecture possible pour le circuit dédié est illustrée à la figure 4.2.

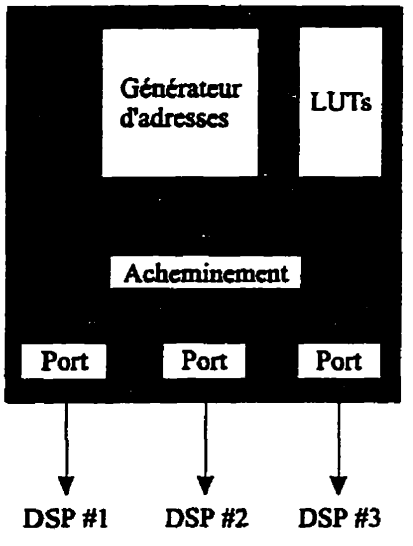


Figure 4.2 : Architecture du circuit dédié.

Afin de connaître les performances de ce système, un circuit dédié conforme à l'architecture de la figure 4.2 a été décrit en langage VHDL et simulé en utilisant la librairie Logic Lib de

Synopsys (Synopsys, 1995) afin de simuler le comportement des trois processeurs. Cette librairie permet de simuler le comportement des trois processeurs, en exécutant les instructions fournies sous forme de trois programmes décrits en langage PDL (semblable au langage C). Le système entier peut ensuite être simulé en utilisant le simulateur VSS de Synopsys. Une fois la simulation satisfaisante, on synthétise le circuit dédié, et on refait la simulation complète du système en utilisant le circuit synthétisé, afin de déterminer si les délais introduits par la synthèse empêchent d'obtenir la fonctionalité originale du circuit. Cette simulation nous a permis d'arriver à la conclusion que le goulot d'étranglement de cet algorithme est la perte de temps due aux communications. Le tableau 4.1 montre la cédule des événements sur chacun des trois processeurs. D'après ce tableau, il faut 29 cycles machine (25 MHz) pour terminer la transformation d'un point (862,068 points par seconde). Sur les 87 cycles disponibles pour faire des calculs (3 DSPs x 29 cycles), seulement 19.5% (17 cycles) sont utilisés par le CPU, 41.4% (36 cycles) par les ports de communication et 39.1% (34 cycles) sont inutilisés.

Afin de diminuer le nombre de cycles inutilisés, il est possible de "pipeliner" les opérations. Ainsi, à partir du cycle 21, le calcul d'un nouveau point peut être commencé. Le pourcentage de cycle inutilisé est maintenant de seulement 11.5% (10 cycles), celui utilisé par le CPU est de 29.9% (26 cycles) et finalement celui utilisé par les ports de communication 58.6% (51 cycles).

Il faut donc maintenant 21 cycles machine pour transformer un point, plus 8 cycles supplémentaires pour le premier point, ce qui correspond à 1,190,476 points par seconde.

L'objectif de 1 million de points par seconde peut donc être atteint avec ce système. Néanmoins, la grande capacité de calcul disponible laissait espérer de meilleurs résultats. Le grand nombre de cycles utilisés pour les communications fait baisser le taux d'utilisation de cette capacité de calcul.

La conception de ce système a donc soulevé une série de questions: Est-il possible de concevoir un système qui utilise plus efficacement sa capacité de calcul? Est-il possible de diminuer le nombre de communications? Est-il possible d'éviter la conception d'un circuit dédié dont la fonction est très limitée?

C'est pour ces raisons que les systèmes des deux sections suivantes ont été développés. Il s'agit dans les deux cas d'un processeur qui fait la totalité des calculs, et qui accède lui même aux LUTs. Puisque tous les points fournis par la caméra sont indépendants, une carte telle que la carte *Blazer* peut ainsi être utilisée, mais pour calculer trois points en parallèle au lieu de travailler sur la même transformation.

Tableau 4.1 : Ordonnancement des opérations sur les trois processeurs.

1 cycle machine = 2 cycles d'horloge

	DSP #1		DSP #2		DSP #3	
	CPU	Ports de comm.	CPU	Ports de comm.	CPU	Ports de comm.
0	i et j → ASIC					
1	calcul de p'	transmission de i et j				
2						
3						
4						
5	calcul de p' - P_n					
6	$G = (p' - P_n)^{-1}$					
7						
8		réception de Z_s				réception de X_s
9						
10						
11						
12						
13		réception de Z_n				réception de X_n
14	G → DSP #3					
15					réception de G	
16						
17						
18	$T = G * Z_s - Z_n$			réception de cos(ϕ)		réception de sin(ϕ)
19	T → DSP #2					
20	T → DSP #3		réception de T		$X = G * X_s - X_n$	
21					réception de T	
22						
23				réception de H_z		réception de H_y
24						
25						
26						
27						
28			$Z = T * \cos(\phi) + H_z$		$Y = T * \sin(\phi) + H_y$	

4.2 Implantation logicielle sur un TMS320C40

4.2.1 Description générale

Comme il a été montré à la section 3.7 du chapitre précédent, l'arithmétique à virgule flottante offre l'avantage de fournir des résultats très précis. La solution étudiée à la section 4.1 fournit des résultats un peu décevant vu la grande capacité de calcul disponible. Le grand nombre de cycles utilisés à des fins de communication fait baisser le taux d'utilisation de cette capacité de calcul.

Ainsi, pour éviter les communications, la solution logicielle sur un seul processeur TMS320C40 de la compagnie Texas Instruments est envisagée dans cette section.

Ce DSP ("Digital Signal Processor") est un processeur à virgule flottante de 32 bits d'usage général (Texas Instruments, 1991). Il a été choisi pour ses unités arithmétiques à virgule flottante, et sa grande popularité dans le domaine de la vision par ordinateur. Son architecture interne est illustrée à la figure 4.3.

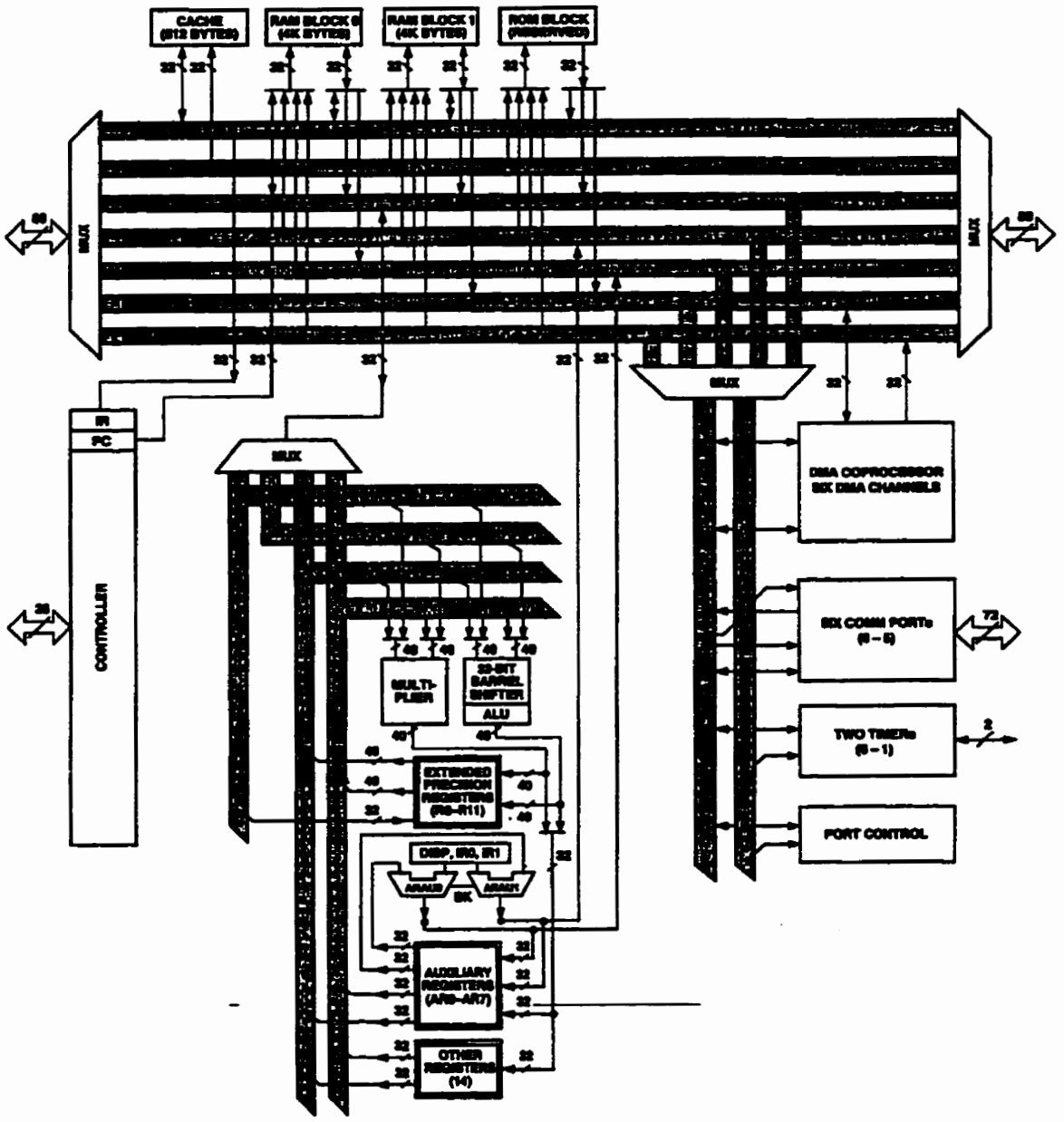


Figure 4.3 : Architecture du TMS320C40

Les principales caractéristiques de l'architecture de ce processeur, sont :

- Deux bus externes ("local bus" et "global bus") qui permettent l'accès à près de 4Goctets de mémoire externe;

- Deux blocs de mémoire RAM internes qui peuvent contenir un total de 8 Koctets d'instruction ou de données;
- Une mémoire cache d'instructions de 512 octets de type LRU ("Least Recently Used");
- 26 registres, dont 12 registres de 40 bits (R0-R11), 8 registres auxiliaires (AR0-AR7), 2 registres d'index (IR0, IR1), etc.;
- Six ports de communication bidirectionnels asynchrones pouvant communiquer à un taux de 160 Mbits par seconde;
- Un DMA ("Direct Memory Access") qui peut lire et écrire dans toutes les adresses disponibles dans le processeur (mémoire, ports de communication, etc.) sans l'intervention du CPU ("Central Processing Unit");
- Un CPU qui dispose d'un multiplieur en parallèle avec l'unité arithmétique et logique, ce qui permet d'effectuer une multiplication et une opération sur l'UAL (Unité Arithmétique et Logique) dans le même cycle.

Adressage

Pour permettre l'accès aux données et aux instructions dans la mémoire et dans les registres, ce processeur permet cinq types d'adressage :

- Registre
- Direct
- Indirect
- Immédiat

• Relatif au PC

L'adressage Direct (syntaxe : @expr) permet d'accéder à une donnée en concaténant les 16 bits les moins significatifs du pointeur de page (DP) avec les 16 bits les moins significatifs du mot d'instruction.

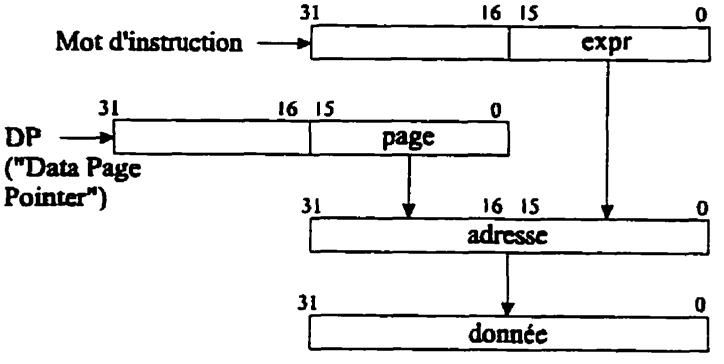


Figure 4.4 : Adressage direct avec le TMS320C40

L'exemple suivant montre l'utilisation de l'adressage direct:

```
ADDI        @0BCDEh, R7
```

Avant l'instruction:
```
DP = 108Ah
R7 = 11h
Donnée à 108A BCDEh = 1234 5678h
```

Après l'instruction:
```
DP = 108Ah
R7 = 1234 5689h
Donnée à 108A BCDEh = 1234 5678h
```

Notez que ce type d'adressage est à éviter dans la partie critique d'un programme. En effet, il faut jusqu'à trois cycles machine (1 cycle machine = 2 cycle d'horloge) pour obtenir la donnée.

L'adressage Indirect (syntaxe : *expr) est utilisé pour spécifier l'adresse d'une donnée en mémoire par le contenu d'un registre auxiliaire, d'un déplacement (8 bits) et d'un registre

d'index. L'exemple suivant montre l'utilisation de l'adressage indirect:

```
STF          R3,*AR4++(IR1)
```

Avant l'instruction:

```
R3 = 057B40 0000h = 6.28125e+01
AR4 = 80 9900h
IR1 = 10h
Donnée à 80 9900h = 70C 8000h
                  = 1.4050e+02
```

Après l'instruction:

```
R3 = 057B40 0000h = 6.28125e+01
AR4 = 80 9910h
IR1 = 10h
Donnée à 80 9900h = 057B40 0000h
                  = 6.28125e+01
```

Le tableau 4.2 donne la syntaxe des principales méthodes de déplacement avec l'adressage indirect.

Tableau 4.2 : Adressage indirect avec déplacement.

`*+ARn(dépl)`	adr = ARn + dépl	Pré-déplacement (addition)
`*-ARn(dépl)`	adr = ARn - dépl	Pré-déplacement (soustraction)
`*++ARn(dépl)`	adr = ARn + dépl ARn = ARn + dépl	Pré-déplacement (addition et modification)
`*--ARn(dépl)`	adr = ARn - dépl ARn = ARn - dépl	Pré-déplacement (soustraction et modification)
`*ARn++(dépl)`	adr = ARn ARn = ARn + dépl	Post-déplacement (addition et modification)
`*ARn--(dépl)`	adr = ARn ARn =ARn - dépl	Post-déplacement (soustraction et modification)
`*ARn++(dépl)%`	adr = ARn ARn = circ(ARn + dépl)	Post-déplacement (addition avec modification circulaire)
`*ARn--(dépl)%`	adr = ARn ARn = circ(ARn - dépl)	Post-déplacement (soustraction avec modification circulaire)

ARn = registres auxilliaires (AR0- AR7)
dépl = déplacement (5 bits ou 8 bits) ou registres d'indexage (IR0 et IR1)

Ce type d'adressage est très intéressant puisqu'il nous permet de lire ou écrire une donnée en mémoire externe en un seul cycle si la mémoire est assez rapide (SRAM de 17 ns dans le cas d'un processeur de 50 Mhz).

L'adressage immédiat (syntaxe : expr) permet de fournir une donnée de 16 bits directement

avec l'instruction. Cette valeur peut être un entier ou un nombre à virgule flottante signé ou non. L'exemple suivant montre l'utilisation de l'adressage immédiat :

```
LDI    0FFFFh,R0
Avant l'instruction:          Après l'instruction:
R0 = 0h                       R0 = 00 0000 FFFFh
```

L'adressage "Relatif au PC" est utilisé pour effectuer des branchements conditionnels. L'utilisateur dispose de dix-neuf types de conditions qui dépendent des "flags" du processeur. Par exemple, un branchement "si égal à zéro" aurait comme instruction :

```
BZ     R0
Avant l'instruction:          Après l'instruction:
PC = 2B00h                    PC = 3FF00h
R0 = 0003FF00h                R0 = 0003FF00h
```

Ce type d'adressage n'est pas très performant. Il faut quatre cycles machine pour effectuer un branchement.

Ports de communication

Les six ports de communication asynchrone du TMS320C40 sont bidirectionnels, et chacun permet un transfert de données allant jusqu'à 160 Mbits par seconde. L'avantage d'utiliser ces ports de communication est le parallélisme qui existe entre les ports et le CPU. En effet, un port de communication est perçu par le CPU comme étant une adresse dans sa mémoire. Pour lire un mot reçu par un port de communication, le CPU vérifie qu'un mot est arrivé en lisant le registre d'état, puis va lire ce mot à l'adresse qui correspond au registre de lecture de ce port. Par exemple, le groupe d'instructions assembleur suivant correspond à la lecture du port de communication #4 d'un TMS320C40 :

```
LDA   100080H,AR7              ** L'adresse du registre d'état est mis
                               ** dans le registre AR7.
      LDI   1E00H,R0
L4:
      TSTB  R0,*AR7            ** Le contenu du registre d'état est testé
                               ** pour déterminer si une valeur est
                               ** arrivée dans le port.
      BZ    L4                 ** Si aucune valeur n'est arrivée,
                               ** brancher à L4.
      LDI   *+AR7(1),R1        ** Sinon, mettre le contenu du registre de
                               ** lecture dans le registre R1.
```

Pour envoyer un mot, le CPU devrait normalement vérifier qu'au moins un des sept FIFOs ("First In First Out") du port soit libre en lisant le registre d'état, puis écrire ce mot à l'adresse qui correspond au registre d'écriture de ce port. Dans le cas de l'algorithme FIT, le débit de données qui seront écrites dans le port de communication est petit (3 données par exécution). Il est donc possible d'écrire directement les données dans le port sans se soucier du débordement des FIFOs. Les instructions suivantes sont celles d'une écriture dans le port de communication #1 du TMS320C40.

```
LDA   100052H,AR7        ** Mettre l'adresse du registre d'écriture dans
                         ** le registre AR7.
STF   R3,*AR7            ** Écrire le contenu du registre R3 dans
                         ** l'adresse contenu dans AR7.
```

Une fois cette dernière instruction exécutée, le CPU peut continuer l'exécution de son programme sans s'occuper de la communication. En effet, celle-ci est orchestrée par les ports qui communiquent ensemble sans aucune intervention de la part des deux CPUs.

Le désavantage du port de communication par rapport à une communication à mémoire partagée est sa vitesse. Il faudra au minimum 5 cycles machine pour transmettre un mot de 32 bits, puisque les ports envoient les données octet par octet. À ces cycles devront être

ajoutés ceux qui sont dus au protocole de synchronisation (au maximum 2 cycles) entre les deux ports qui communiquent ensemble, ceux qui sont dus au changement de direction des ports, etc.

Unité Centrale

Le CPU possède une propriété très intéressante pour cette application. Il est possible d'exécuter certaines instructions en parallèle. L'unité de multiplication, l'UAL (Unité Arithmétique et Logique) et certaines instructions de communication (LDF, STF, etc.) peuvent être exécutées dans le même cycle machine. La seule contrainte qui doit être respectée, c'est de ne pas utiliser les mêmes registres pour les deux instructions exécutées en parallèle.

Par exemple, si on veut faire une multiplication et une soustraction en parallèle sur des nombres à virgule flottante, l'instruction serait :

```
        MPYF3       *+AR0(IR1),R4,R0
||      SUBF3 *+AR3(IR1),R3,R2
```

4.2.2 Algorithme FIT sur TMS320C40

L'architecture qui est proposée est basée sur un TMS320C40 qui accède à une mémoire dans laquelle les tables de références sont gardées. Les triplets (p,i,j), qui sont fournis par la caméra autosynchronisée, sont transférés au processeur par un port de communication. Comme il est montré à la figure 3.1 du chapitre précédent, il est possible d'accéder directement aux résultats préliminaires dans les LUTs à partir de (p,i,j). Les calculs

nécessaires pour obtenir les points (x,y,z) seront ensuite effectués par le CPU qui transférera les résultats par un autre port de communication. La figure 4.5 montre le schéma du système.

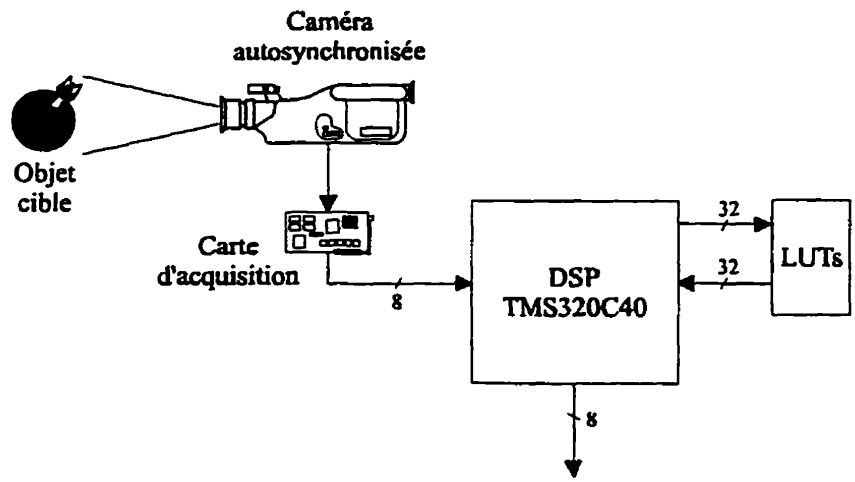


Figure 4.5 : Architecture du système basé sur un TMS320C40

Dans un premier temps, le programme a été décrit en langage C (voir annexe C). La performance de la boucle de calcul (sans communication) est de seulement 609,756 points par seconde. Ce résultat décevant s'explique en partie par la difficulté qu'a le compilateur à paralléliser les calculs de l'algorithme. En effet, la difficulté réside dans le fait que les deux opérations qui sont éffectuées en parallèle ne doivent pas utiliser les mêmes registres. L'algorithme FIT est basé surtout sur des multiplications-accumulations, ce qui implique que le résultat de la multiplication doit être additionné par la suite. Il est donc évident qu'un registre devra être partagé, ce qui exclut l'exécution parallèle de la multiplication-accumulation. Il faut donc changer l'ordre des calculs pour que chaque multiplication d'une équation soit parallélisée avec une addition d'une autre équation. Ce type de problème est trop complexe pour être résolu par le compilateur.

Un autre problème qu'a le compilateur est de gérer l'utilisation des registres. Comme il a été

dit plus haut, l'utilisation de l'adressage direct doit être évité dans la partie critique du programme, puisqu'il faut plusieurs cycles pour exécuter l'instruction. Ainsi, il faut gérer les huit registres auxiliaires de façon très rigoureuse afin d'utiliser l'adressage indirect pour accéder aux LUTs et aux registres des ports de communication. L'algorithme demande l'accès à neuf tables et à deux ports de communication. Le TMS320C40 ne possèdant que huit registres auxiliaires, certaines adresses doivent être mises dans des registres "ordinaires" (R0-R11) afin de les copier dans des registres auxiliaires avant leur utilisation.

Pour résoudre ces problèmes, l'étape suivante fut le développement d'un programme optimisé au niveau du langage assembleur (voir annexe G). Au début du programme, tous les registres auxiliaires sont assignés à une LUT.

```
LDA     @CONST+0,AR0        ** adresse de Xg[0]
LDA     @CONST+1,AR1        ** adresse de Xo[0]
  .
  .
  .
LDA     @CONST+9,AR7        ** adresse de port4_sts
LDI     AR7,R11             ** copie de l'adresse de port4_sts dans R11
```

Notez que les constantes dénotées `CONST+n` sont les adresses de la première donnée de la LUT. Elles sont définies à la fin du programme (annexe G).

Ensuite, les triplet (p,i,j) sont lus dans le port de communication (voir section 4.2.1 - Ports de communication).

La troisième partie du programme, est la boucle de calcul qui doit être optimisée en séparant les multiplications et les additions de chaque équation afin de les exécuter de façon parallèle.

```
    MPYF3   *+AR2(IR1),R4,R3   ** Zg(i) * G(p) -> R3

    MPYF3   *+AR0(IR1),R4,R0   ** Xg(i) * G(p) -> R0
||  SUBF3   *+AR3(IR1),R3,R2   ** Zg(i) * G(p) - Zo(i) -> R2

    MPYF3   *+AR4(IR0),R2,R1   ** [Zg(i)*G(p)-Zo(i)]*sin(j) -> R1
||  SUBF3   *+AR1(IR1),R0,R3   ** X = [Xg(i) * G(p)] - Xo(i) -> R3

    MPYF3   *+AR6(IR0),R2,R0   ** [Zg(i)*G(p)-Zo(i)]*cos(j)->R0
||  ADDF3   *+AR5(IR0),R1,R2   ** Y=[Zg(i)*G(p)-Zo(i)]*sin(j)+Hy(j)-> R2

    ADDF3   *+AR7(IR0),R0,R1   ** Z=[Zg(i)*G(p)-Zo(i)]*cos(j)+Hz(j)-> R1
```

Selon les données de Texas Instruments, l'exécution de la boucle de calcul de l'algorithme nécessiterait 10 cycles d'horloge. En pratique, il faut 18 cycles d'horloge pour exécuter ces 5 instructions. La différence entre les deux provient du temps d'accès à la mémoire. En effet, chaque instruction parallèle fait deux accès à la mémoire, accès qui nécessitent chacun environ 20 ns. En plus, la multiplication et l'addition sur ces données doivent être traitées dans le même cycle machine (40 ns). Il faudra donc au moins deux cycles machine pour chacune des trois instructions parallèles.

Le chapitre 5 traite en profondeur la performance obtenue par ce système. Néanmoins, pour faciliter la lecture, une partie de ces résultats est reproduite dans le tableau 4.3.

Tableau 4.3: Temps d'exécution des parties du programme optimisé sur TMS320C40.

Communication	44 cycles d'horloge
Boucle de calcul	18 cycles d'horloge
Contrôle	28 cycles d'horloge

D'après ces résultats, il faut au total 90 cycles d'horloge (555,555 points par seconde incluant les communications) pour faire la transformation d'un triplet (p,i,j) en un point de l'espace

cartésien.

Toutefois, il faut remarquer que contrairement à ce qu'y avait été prévu, le goulot d'étranglement du système provient des communications. En fait, des 44 cycles d'horloge alloués aux communications, 93% (41/44) proviennent de la routine de vérification du port de communication.

Afin de diminuer l'impact des communications sur l'exécution de l'algorithme, on peut faire l'hypothèse que le débit de données qui arrivent de la caméra peut être ajusté afin de synchroniser la sortie de la carte d'acquisition avec le processeur et ainsi garantir qu'une donnée se trouve dans le port de communication au moment ou l'algorithme en a besoin. Cette hypothèse permet d'éliminer la routine de vérification du port et de ce fait, couper 41 cycles d'horloge à l'exécution de l'algorithme.

La performance du système devient ainsi 49 cycles d'horloge par transformation, ce qui représente un débit total de 1,010,408 points par seconde. On est donc parvenu à atteindre un des objectifs principaux de ce travail qui est de 1 million de transformations par seconde. Néanmoins, ces résultats sont fondés sur une hypothèse faite sur une caméra et sur une carte d'acquisition qui sont toujours sur les tables à dessin. La section prochaine discute donc d'une autre version logicielle de l'algorithme, mais qui utilise un processeur SHARC au lieu du TMS320C40.

4.3 Implantation logicielle sur un ADSP-21060

4.3.1 Description générale

La section précédente a permis de déterminer qu'un des problèmes clés de l'implantation de cet algorithme sur un processeur, c'est le nombre de cycles nécessaires pour gérer les ports de communication. Pour atteindre les performances voulues avec le TMS320C40 de Texas Instruments, il aura fallu faire une hypothèse qui ne dépend pas de l'implantation de l'algorithme, mais de la caméra et de la carte d'acquisition. Il faut donc trouver une solution qui permette d'éliminer cette hypothèse. L'architecture qui est étudiée dans cette section est basée sur un processeur ADSP-21060 de Analog Devices, nommé SHARC (Super Harvard Architecture Computer), qui accède aux données dans les LUTs. Ce DSP fait partie de la famille des processeurs 32 bits à virgule flottante d'usage général (Analog Devices, 1995). La figure 4.6 montre l'architecture générale du SHARC.

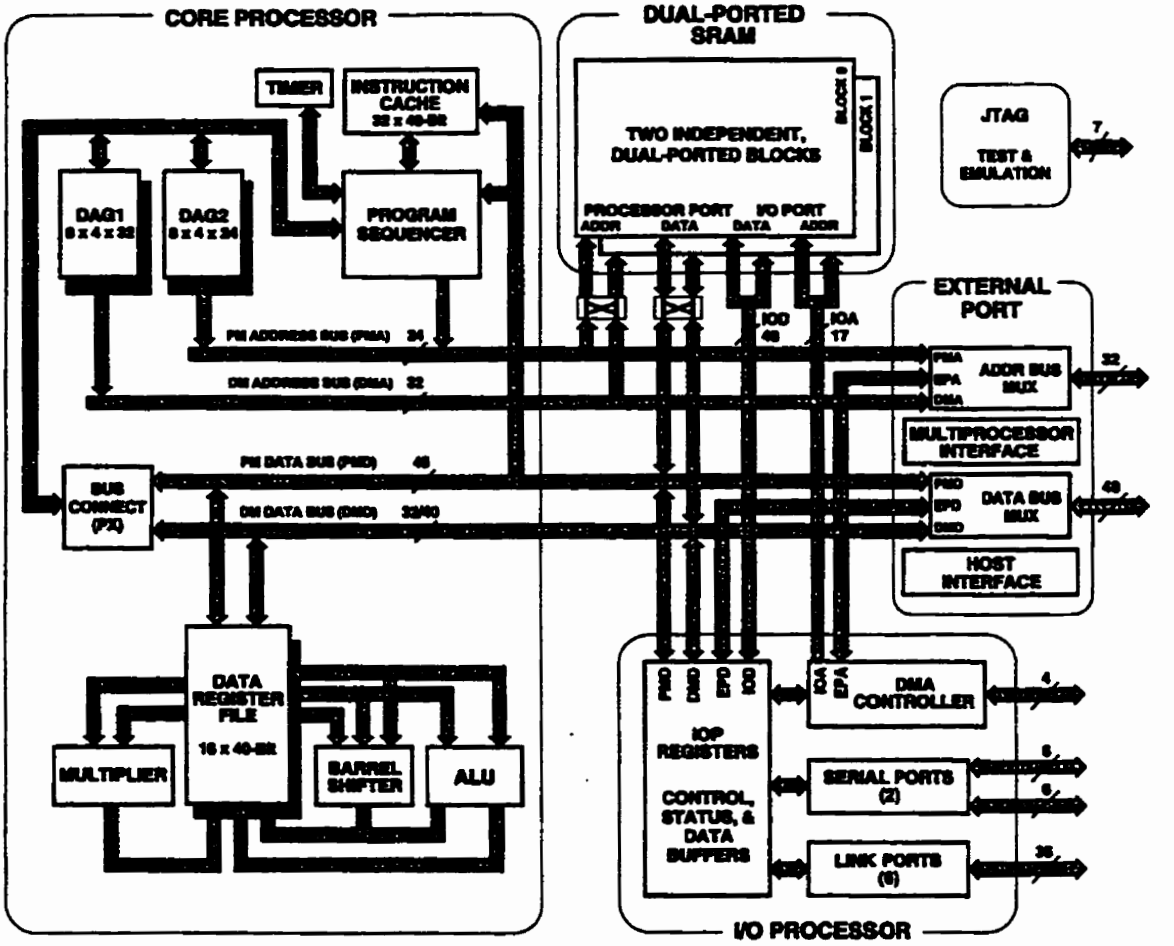


Figure 4.6 : Architecture du ADSP-21060

Les principales caractéristiques de ce processeur sont les suivantes :

- Un bus externe qui permet d'accéder à 4 Giga mots de 48 bits;
- Deux bus internes : dm ("Data Memory") et pm ("Program Memory");
- Deux blocs de mémoires SRAM internes de 2 Mbits chacun;
- Une mémoire cache pouvant contenir 32 instructions de 48 bits;
- 80 registres de données ou d'adresses

- 16 registres de données de 40 bits (R0-R15 ou F0-F15);
- 16 registres d'index (I0-I15);
- 16 registres "Modify" (M0-M15);
- 16 registres de base et 16 registres utilisés pour l'adressage circulaire;

• 6 ports de communication asynchrone ("Link Ports") de 40 Moctets par seconde;

• 2 ports de communication synchrone ("Serial Port") de 40 Mbits par seconde;

• Un DMA ("Direct Memory Access") qui permet de transférer des blocs de données;

• Un CPU ("Central Processing Unit") qui permet de faire jusqu'à cinq opérations (une multiplication, une addition, une soustraction et deux accès mémoire) dans le même cycle.

Adressage

Ce processeur permet trois types d'adressage regroupés dans deux familles:

- Direct (absolu, relatif au PC)
- Indirect

L'adressage absolu permet d'accéder (lecture ou écriture) à une adresse qui ne changera pas lors de l'exécution du programme. Les deux exemples suivants montrent la syntaxe d'un adressage absolu.

1) `dm(0x000015F0) = 35;  ! La valeur 35 sera écrite à l'adresse 0x000015F0`

2) `if ne jump 0x00200001;  ! Si différent aller à l'adresse 0x00200001`

L'adressage "relatif au PC" permet de faire un branchement conditionnel ou d'exécuter une sous-routine. Il existe 34 conditions pour déterminer s'il y a branchement ou non. Les exemples suivants montrent les deux types d'adressage relatif au PC.

1) `call(pc,10);  ! On branche à la sous-routine qui est à pc + 10`

2) `do (pc,4) until LCE;  ! On recommence les 4 instructions suivantes jusqu'à ce que le compteur de boucles soit nul ("Loop Counter Expired")`

L'adressage indirect se divise en quatre sections :

Post-modification avec registre M et mise à jour du registre I

1) `f5 = pm(i9,m12);`

2) `dm(i0,m3) = r3;`

Pré-modification avec registre M sans mise à jour du registre I;

1) `r1 = pm(m10,i15);`

2) `jump(m13,i11);`

Post-modification avec valeur immédiate et mise à jour du registre I;

1) `f15=dm(i0,6);`

2) `if av r1=pm(i15,0x11);  ! av : ALU overflow`

Pré-modification avec valeur immédiate sans mise à jour du registre I.

1) `if av r1=pm(0x11,i15);`

2) `dm(127,i5) = laddr;`

Note : Contrairement au TMS320C40, tous les types d'adressage du SHARC ne nécessitent qu'un seul cycle d'horloge.

Ports de communication

Le SHARC possède deux types de ports de communication bidirectionnels : les "Link Ports" et les "Serial Ports". Il existe plusieurs différences entre les deux types :

- Les "Link Ports" sont asynchrones, tandis que les ports sériel (SPORT) sont synchrones;
- Le SHARC a six "Link Port" et seulement deux SPORTs;
- Les ports de communication de type "Link Port" peuvent communiquer à un taux de 40 Moctets par seconde tandis que les SPORTs à un taux de 40 Mbits par seconde;
- Les "link ports" utilisent une communication de 4 bits parallèles contrairement aux ports sériels.

Les ports de communication sont vus par le CPU comme étant des regitres en mémoire. Avant d'utiliser les ports de communication, il faut initialiser ceux qui seront utilisés. Cette initialisation ne se fait qu'une seule fois au début de l'exécution et ne fait donc pas partie de la boucle critique de l'exécution de l'algorithme.

L'initialisation des "link ports" se fait en trois étapes. Premièrement il faut fixer la vitesse de la communication. Deux choix s'offrent à l'utilisateur (Analog Devices, 1997, p.9-27) : communication à la vitesse de l'horloge ou bien à deux fois la vitesse de l'horloge. Ceci se fait de la façon suivante :

```
R0=0x0000C000;     ! Les "link ports" 2 et 3 communiqueront à deux
dm(0x00C7)=R0;     ! fois la vitesse de l'horloge.
```

Deuxièmement, il faut assigner les "link ports" avec un "buffer" (Analog Devices, 1997, p.9-10) puisqu'il est possible d'assigner n'importe quel "link ports" avec n'importe quel "link buffer". Cette opération est faite comme suit :

```
R0=0x0003F03F;      ! Les ports 2 et 3 sont assignés au buffer 2 et 3
dm(0x00C8)=R0;
```

Troisièmement, il faut décider de la direction des communications pour chaque port (Analog Devices, 1997, p. 9-4) de la façon suivante :

```
R0=0x00009100;      ! Le "link port" 2 est mis en mode réception
dm(0x00C6)=R0;      ! tandis que le 3 est en mode transmission
```

Après avoir vérifié le registre d'état pour déterminer si le FIFO ("First-in First-Out") du port est plein ou non, un mot peut être envoyé en l'écrivant dans le registre d'écriture du port. Dans le cas de l'algorithme FIT, le débit de données à envoyer par le port de communication étant faible, il n'est pas nécessaire de vérifier le contenu du registre d'état avant d'écrire le mot dans le registre du port. L'exemple suivant montre la syntaxe d'une écriture du registre f0 dans un port "link port" du SHARC :

```
dm(0x00f2) = f0;
```

De la même façon, pour lire une donnée qui est arrivée par le port, on vérifie le registre d'état du port pour s'assurer qu'une donnée est présente, puis on lit le registre de lecture du port. L'exemple suivant montre la syntaxe d'une lecture dans un port sériel (SPORT1) :

```
i0=dm(0x00C2);      ! Le programme attendra ici jusqu'à ce qu'il y
                    ! ait une donnée dans le port.
```

Unité Centrale

Comme il est possible de le voir à la figure 4.6, le CPU du processeur est composé de trois

unités opératives en parallèle : un multiplieur, un UAL et un décaleur. Ce parallélisme permet d'exécuter jusqu'à 5 instructions dans le même cycle. Par exemple, l'instruction suivante fait une multiplication, une addition, une soustraction et deux accès mémoire dans le même cycle d'horloge.

```
R11=R1*R7, R3=R9+R14, R9=R9-R14, dm(I2, M0)=R13, R7=pm(I8,M8);
```

Comme dans le cas du TMS320C40, certaines restrictions doivent être respectées. Par exemple, les opérandes de l'addition et celles de la soustraction doivent être dans les mêmes registres, qui doivent être différents de ceux utilisés pour la multiplication.

4.3.2 Algorithme FIT sur ADSP-21060

Une architecture basée sur le SHARC est illustrée à la figure 4.7. Les données (p,i,j) fournies par la caméra autosynchronisée seront envoyées par le port de communication "link port" #2. Le processeur, après avoir fait la transformation des données en des points de l'espace cartésien, sortira ces résultats par un autre port de communication "link port" #3.

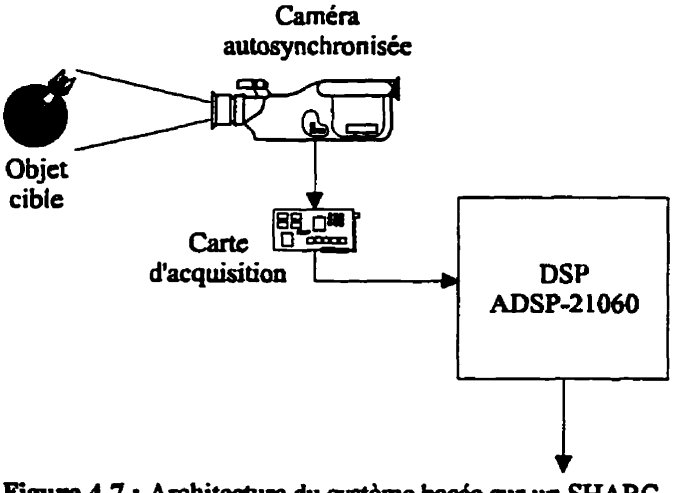


Figure 4.7 : Architecture du système basée sur un SHARC

Le fait que les LUTs ne soient pas représentées sur la figure 4.7 n'indique pas que le système n'en utilise pas, mais plutôt que ce processeur a l'avantage de posséder une mémoire interne assez grande afin de mettre toutes les LUTs à l'intérieur du DSP. En effet, dans le cas où i et j sont sur 9 bits et que p soit sur 16 bits, il faudra 8 tables de 512 mots et une table de 65,536 mots de 32 bits, ce qui fait un total de 69,632 mots. Les LUT pourront donc aisément être mises dans la mémoire interne qui peut contenir jusqu'à 128,000 mots de 32 bits.

Dans un premier temps, l'algorithme a été décrit en langage C (voir annexe H). La performance ainsi obtenue pour la boucle de calcul sans les communications est de 526,315 points par seconde. Ce mauvais résultat s'explique par les mêmes raisons que celui obtenu avec le TMS320C40 (voir section 4.2.2). En effet, le compilateur a de la difficulté à paralléliser les opérations et à gérer les nombreux registres de façon efficace.

La prochaine étape fut donc de décrire l'algorithme en langage assembleur afin d'obtenir de meilleures performances (voir annexe I).

Le programme débute par l'initialisation des registres qui permettent d'accéder aux LUTs.

```
m0=_Xg;
m1=_Xo;
.
.
.
m7=_Hz;
m8=_G;
```

Contrairement au TMS320C40, le SHARC dispose d'assez de registres afin de ne pas devoir utiliser le même pour accéder à deux tables différentes. On gagne ainsi plus de 4 cycles par

exécution de l'algorithme.

Ensuite, les entrées (*p,i,j*) sont lues dans les ports de communication (voir 4.3.1 - Port de communication).

La partie suivante du programme est la boucle de calcul de l'algorithme. Comme il a été mentionné dans la section précédente, chacune des instructions suivantes ne demande qu'un seul cycle.

```
f4=dm(m8,i0);            ! Récupérer G dans la LUT
f0=dm(m2,i2);            ! Récupérer Zg dans la LUT
f8=f4*f0,f1=dm(m0,i2);   ! Zg * G || récupérer Xg
f9=f1*f4,f12=dm(m3,i2); ! Xg * G || récupérer Zo
f2=f8-f12,f5=dm(m4,i1); ! Zg * G - Zo || récupérer sin
f8=f2*f5,f12=dm(m1,i2); ! (Zg * G -Zo) * sin || récupérer Xo
f0=f9-f12,f6=dm(m6,i1); ! Xg * G - Xo || récupérer cos
f3=f2*f6,f12=dm(m5,i1); ! (Zg * G - Zo) * cos || récupérer Hy
f1=f8+f12,f4=dm(m7,i1); ! [(Zg * G - Zo) * sin] + Hy || récupérer Hz
f2=f3+f4;                ! [(Zg * G - Zo) * cos] + Hz
```

Il faut donc 10 cycles d'horloge de 40 MHz pour l'exécution de cette partie du code, ce qui représente une amélioration de 56% par rapport au code assembleur sur le TMS320C40 (voir sections 4.2.2 et 5.5). Un sommaire des résultats obtenus avec ce système est donné dans le tableau 4.4.

Tableau 4.4 : Temps d'exécution des parties du programme assembleur sur ADSP-21060

Communication	15 cycles d'horloge
Boucle de calcul	10 cycles d'horloge
Contrôle	3 cycles d'horloge

Ce système donne une performance de 1,428,571 transformations par seconde.

Les trois systèmes précédents utilisent une arithmétique à virgule flottante pour obtenir une très bonne précision. Néanmoins, nous avons vu au chapitre 3 qu'il est possible d'utiliser une

arithmétique à virgule fixe en obtenant une erreur relativement faible sur les résultats. L'architecture suivante est une implantation sur un circuit dédié qui utilise une représentation des nombres en virgule fixe sur 20 bits.

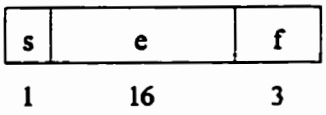
4.4 Implantation matérielle sur un FPGA de Xilinx

Comme il est montré à la section 3.7, l'avantage qu'offre une représentation à virgule flottante est la grande précision des résultats. Cette précision est néanmoins obtenue au dépend de la surface nécessaire à l'implantation des unités fonctionnelles. Une unité arithmétique à virgule fixe (FXU) utilise une configuration matérielle identique à celle qu'utilise une unité arithmétique à virgule flottante (FPU) pour sa partie fractionnaire. La surface additionnelle, nécessaire pour la FPU, vient de la circuiterie nécessaire pour l'alignement des fractions, pour la comparaison des exposants, pour le contrôle du décaleur, ainsi que pour d'autres caractéristiques propres à l'utilisation d'unités à virgule flottante (Cavanagh, 1984).

Vu la taille des unités opératives à virgule flottante, la mise en oeuvre d'une architecture sur un FPGA devra utiliser une représentation des nombres en virgule fixe. La section 3.7 du chapitre précédent détaille l'étude d'une représentation de l'algorithme FIT en virgule fixe. Cette étude permet de déterminer qu'une représentation des nombres sur 24 bits est nécessaire pour obtenir une erreur de 5 dans 100,000 sur les résultats des transformations géométriques. Cependant, grâce à l'accélération par l'utilisation de LUTs, une précision du même ordre peut être obtenue à l'aide de nombres représentés sur moins de bits. La grande précision des nombres pré-calculés dans les tables, ainsi que la diminution de la quantité d'opérations effectuées sur ces nombres, en comparaison d'un calcul normal, permettent d'obtenir avec seulement 20 bits, une perte de précision équivalente à celle obtenue avec une représentation 24 bits.

L'architecture illustrée à la figure 4.8 utilise une représentation des nombres sur 20 bits, dont

le premier bit correspond au signe du nombre, les 16 suivants à la partie entière et les 3 derniers à la partie fractionnaire.



Comme il a été dit au chapitre précédent, l'utilisation de LUTs permet d'éliminer les décaleurs de l'architecture du système, puisque tous les décalages peuvent être faits de façon câblée. De plus, si le volume que la caméra auto-synchronisée doit balayer est modifié, seul le contenu des LUTs doit être modifié.

Ces effets inhérents à l'utilisation de LUTs, ainsi que le gain de performance obtenu, incitent le développement de l'architecture de la figure 4.8[1].

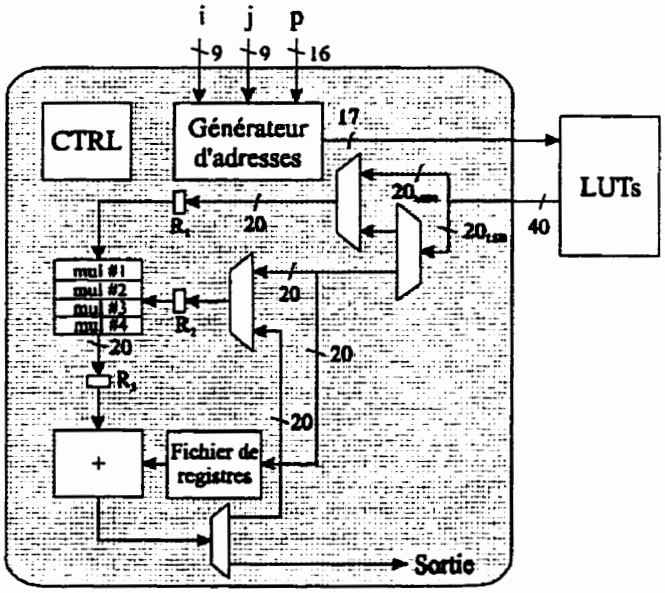


Figure 4.8 : Unité fonctionnelle

---

[1] Le résultat de la multiplication de deux nombres de 20 bits devrait être représenté sur 40 bits. La section 3.7 du chapitre précédent a permis de déterminer qu'il est possible de faire un décalage câblé afin de garder le résultat sur 20 bits.

4.4.1 Mémoire

La mémoire reçoit une adresse en entrée, et retourne la/les valeur(s) qui correspond(ent) à cette adresse. Le bus de sortie a une largeur de 40 bits afin d'obtenir deux valeurs de 20 bits à chaque accès. La mémoire est organisée de la façon suivante :

Tableau 4.5 : Organisation de la mémoire

		Sortie (40 bits)	
		20_{MSB}	20_{LSB}
Adresses (16 bits)	0 @ 32,767	$1/(p - P_{-})$	$1/(p - P_{-})$
	32,768 @ 33,279	$Z_g[i]$	$Z_o[i]$
	33,280 @ 33,791	$X_g[i]$	$X_o[i]$
	33,792 @ 34,303	$\sin[j]$	$H_y[j]$
	34,304 @ 34,815	$\cos[j]$	$H_z[j]$

La table $1/(p - P_{-})$ est repliée en mettant les 32,768 premières valeurs à la place des 20 MSB de la mémoire, et les 32,768 dernières valeurs à la place des 20 LSB.

Dans le cas où p est sur 16 bits et i, j sont sur 9 bits, la taille minimale de cette configuration de la mémoire sera 34,816 adresses * 5 octets/adresses = 170 Koctets. Dans l'éventualité où p, i et j sont sur 16 bits, l'adressage devra être fait sur 18 bits et la mémoire aura une taille de 1.64 Moctets.

4.4.2 Générateur d'adresses

Le générateur d'adresses reçoit les vecteurs d'entrées (p sur 16 bits; i, j sur 9 bits) et génère les adresses qui doivent être accédées en mémoire. Le générateur d'adresses effectue les étapes suivantes :

1) Pour obtenir $1/(p - P_{-})$
   - Si p est plus petit que 32,768
     Concaténer `0` avec p et envoyer à la mémoire;
   - Si p est plus grand que 32,768
     Diviser p par 2 et concaténer `0` avec p et envoyer à la mémoire;
2) Concaténer 80_{16} avec i et envoyer à la mémoire pour obtenir $Z_g[i]$ et $Z_o[i]$;
3) Concaténer 81_{16} avec i et envoyer à la mémoire pour obtenir $X_g[i]$ et $X_o[i]$;
4) Concaténer 82_{16} avec j et envoyer à la mémoire pour obtenir $\sin[j]$ et $H_y[j]$;
5) Concaténer 83_{16} avec j et envoyer à la mémoire pour obtenir $\cos[j]$ et $H_z[j]$.

Pour générer les adresses, il aurait été possible d'additionner une base appropriée (somme de mutiples de 2^{15} ou 2^{9}) à i, j ou p selon le cas. L'utilisation d'une addition au lieu d'une concaténation aurait requis un autre additionneur et aurait généré un délai supplémentaire engendré par celui-ci. La concaténation est donc préférable dans le cadre d'une réalisation matérielle. Il en serait autrement si on procédait à une réalisation logicielle selon cette approche. Il est ainsi possible d'obtenir une paire valeurs de la mémoire en un cycle d'un système opérant à 50 MHz dans le cas d'une mémoire statique rapide.

4.4.3 Ordonnancement des opérations

Avant de déterminer l'ordonnancement des opérations, il faut d'abord spécifer qu'afin d'obtenir des performances acceptables, le multiplieur de la figure 4.8 doit être "pipeliné". En effet, un multiplieur purement combinatoire ajoute un délai considérable sur le flot de données, et il devient généralement le chemin critique du système. Le fait de diviser l'exécution d'une multiplication, et d'isoler chacune des parties résultantes par des registres,

nous permet d'augmenter la fréquence de l'horloge et le débit total du multiplieur, puisque plusieurs multiplications peuvent être actives dans le multiplieur à un cycle donné (voir section 4.4.4).

Le tableau 4.6 résume les étapes de l'algorithme. Les colonnes Mult #1 à Mult #4 correspondent au contenu des étages du multiplieur pipeliné à différents moments du calcul.

Tableau 4.6 : Résumé des étapes de l'algorithme

Cycle	Accès mémoire 20_{mem}	Accès mémoire 20_{mem}	Mult étage #1	Mult étage #2	Mult étage #3	Mult étage #4	Add
1		inv[p]					
2	$Z_{pi}[i]$	$Z_{ci}[i]$					
3	$X_{pi}[i]$	$X_{ci}[i]$	$Z_{pi}[i]$*inv[p]				
4			$X_{pi}[i]$*inv[p]	$Z_{pi}[i]$*inv[p]			
5				$X_{pi}[i]$*inv[p]	$Z_{pi}[i]$*inv[p]		
6					$X_{pi}[i]$*inv[p]	$Z_{pi}[i]$*inv[p]	
7	sin[j]	$H_y[j]$				$X_{pi}[i]$*inv[p]	G=($Z_{pi}[i]$*inv[p])-$Z_{ci}[i]$
8	cos[j]	$H_x[j]$	sin[j]*G				X = ($X_{pi}[i]$*inv[p])-$X_{ci}[i]$
9		inv[p]	cos[j]*G	sin[j]*G			
10	$Z_{pi}[i]$	$Z_{ci}[i]$		cos[j]*G	sin[j]*G		
11	$X_{pi}[i]$	$X_{ci}[i]$	$Z_{pi}[i]$*inv[p]		cos[j]*G	sin[j]*G	
12			$X_{pi}[i]$*inv[p]	$Z_{pi}[i]$*inv[p]		cos[j]*G	Y = (sin[j]*G)+$H_y[j]$
13				$X_{pi}[i]$*inv[p]	$Z_{pi}[i]$*inv[p]		Z = (cos[j]*G)+$H_x[j]$

Le concept de *pipeline* est utilisé à deux niveaux dans ce système. Comme il a été dit plus haut, le premier niveau de pipeline est la division en quatre étages du multiplieur par des registres. Le deuxième niveau consiste à imbriquer le calcul de deux points subséquents à partir du cycle 9 jusqu'au cycle 13. Ainsi la transformation d'un seul triplet (p,i,j) en des points (x,y,z) demande 13 cycles à cette architecture, mais lors de calculs soutenus, il ne faut

que 8 cycles par point, auxquels s'ajoutent 5 cycles de latence. Par exemple, la transformation de 1,000 points sans pipeline nécessite 13,000 cycles, tandis que seulement 8,005 cycles sont nécessaires lorsque le système est pipeliné.

4.4.4 Surface et performance

Afin d'estimer la surface requise pour l'implantation d'une seule unité fonctionnelle, Jeremy Goolsby de la compagnie *Xilinx inc.*, nous a fourni une méthode d'estimation de la complexité des différentes unités opératives (voir tableau 4.7).

Tableau 4.7 : Règle du pouce pour la surface des unités opératives sur un FPGA de Xilinx.

	Surface
Multiplieur (n × n)	n(n-1) CLBs
Additionneur	2 bits par CLB

L'estimé de la surface requise pour l'implantation d'un multiplieur 20×20, est donc de 380 CLBs ("Configurable Logic Block"), tandis que seulement 10 CLBs seront requis pour l'additionneur.

Pour obtenir la performance du multiplieur nous disposons d'une bibliothèque d'unités opératives de la compagnie *Xilinx inc.*. La performance et la surface de chacun des multiplieurs disponibles sont listées dans le tableau 4.8.

Tableau 4.8 : Surface et performance des multiplieurs sur Xilinx

	Surface (CLBs)	Performance (MHz)	Étages de pipeline
8 × 8	54	85	3
10 × 10	94	78	3
12 × 12	122	69	4
16 × 16	216	56	4

En se basant sur ces données, on peut obtenir un estimé de la performance du multiplieur 20×20 (voir fig. 4.9).

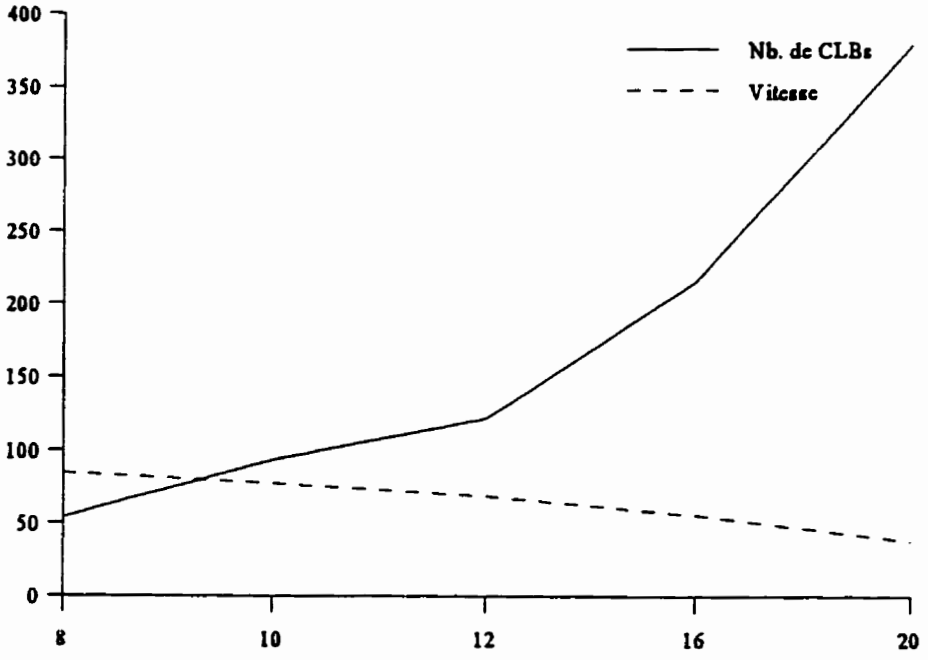


Figure 4.9 : Variation de la taille et de la vitesse du multiplieur en fonction du nombre de bits.

D'après ces données, il est possible d'effectuer une multiplication sur deux nombres de 20 bits à 35 MHz. Sachant que le multiplieur est l'unité la plus lente de cette architecture, le cycle

de l'horloge peut donc être fixé à 30 MHz.

Avec cette approche, le nombre de transformations par seconde qu'il est possible de calculer est de :

$$\left(30\times10^{6}\ \frac{cycles}{sec} \div 8\frac{cycles}{point}\right)+5 \approx 3.75\times10^{6}\ \frac{points}{sec}$$

Nous avons vu au tableau 4.7 qu'il est possible d'estimer la taille des unités opératives grâce à la règle du pouce fournie par *Xilinx inc.*. Néanmoins pour connaître la complexité des autres unités tels que le générateur d'adresses, le fichier de registres, les multiplexeurs et l'unité de contrôle, une autre méthode doit être utilisée.

Toutes ces unités ont donc été décrites en langage VHDL, simulées et synthétisées en utilisant la librairie de Xilinx. La surface obtenue pour chacune d'elles est donnée dans le tableau 4.9.

Tableau 4.9 : Surface des unités de l'architecture

	Nb. de CLBs	Total
Générateur d'adresses	22	22
Unité de contrôle	153	153
4 multiplexeurs	10/multiplexeur	40
7 registres de 20 bits	20/registre	140
		355

Il faudra donc environ 400 CLBs pour le multiplieur, 10 CLBs pour l'additionneur et 355 CLBs pour le reste du système. L'unité fonctionnelle a donc besoin de 765 CLBs, et de 111 plots d'entrées/sorties : 34 plots d'entrées pour (p,i,j), 17 plots de sorties pour le bus

d'adresses vers la mémoire, 40 d'entrées pour les données venant de la mémoire, et 20 autres pour la sortie des résultats. Il est ainsi possible d'implanter le système dans un XC4020E possèdant 784 CLBs et 224 plots d'entrées/sorties alloués à l'utilisateur (Xilinx, 1996).

Ce chapitre avait pour but de définir différents systèmes pouvant calculer la transformation de 1 million de triplets (p,i,j) par seconde. Le système basé sur une version logicielle de l'algorithme implanté sur un TMS320C40 a atteint ce but, mais en faisant des hypothèses sur la conception de la caméra, tandis que les autres l'ont atteint sans faire d'hypothèse. Le chapitre suivant compare les performances obtenues par ces systèmes, et discute des coûts de chacun d'eux versus les performances qu'ils fournissent, afin de faire ressortir le système le plus intéressant pour l'application de la caméra auto-synchronisée.

Chapitre 5 : Résultats et discussions

Le chapitre précédent étudie l'architecture de différents systèmes qui transforment des triplets (p,i,j) fournis par la caméra auto-synchronisée, en des points (x,y,z) du plan cartésien. Afin de déterminer lequel de ces systèmes fournit la meilleure relation coût-performance, ce chapitre a deux objectifs : récapituler les performances obtenues par chacun d'eux et fournir une brève discussion de leurs coûts. L'intention de cette discussion n'est pas d'être une étude exhaustive des produits disponibles sur le marché, mais plutôt de fournir un ordre de grandeur afin de conclure si un système est plus avantageux qu'un autre[1].

5.1 Système mixte logiciel-matériel

Le système mixte logiciel-matériel qui a été décrit à section 4.1 du chapitre précédent, est formé d'une carte commerciale (carte *Blazer*) possédant trois DSP TMS320C40, et d'une carte qui dispose d'un circuit FPGA programmable par l'utilisateur.

Cette seconde carte peut être elle aussi achetée. La compagnie Mirotech a développé la carte XT10-4MCP possédant deux Xilinx XC4013, et des ports de communication compatibles avec ceux des TMS320C40. Néanmoins, cette carte ne peut être utilisée sans une autre carte pour la supporter. La carte ZP10, développée par Mirotech, possède deux emplacements pour des modules TIM (Texas Instruments Module). Cette configuration demande donc la carte Blazer, qui coûte à elle seule 10,000$ US, un module XT10-4MCP au prix de 8,000$, et la carte ZP10 au coût de 8,000$. Afin de diminuer le coût de ce système, il est possible de

---

[1] Pour avoir plus de détails sur les cartes mentionnées dans ce chapitre, se référer à l'annexe J.

remplacer la carte Blazer par la carte F5MS1 PCI de la compagnie Spectrum Signal Processing. Cette carte dispose d'un processeur TMS320C40, et de trois emplacements qui peuvent recevoir trois modules TIM. De plus, cette compagnie a conçu le module MDC40SS2 qui dispose d'un TMS320C40 avec 1 Moctets de SRAM. Deux de ces modules peuvent être mis sur la carte F5MS1 PCI, couplés avec un module XT10-4MCP de Mirotech. Le coût des deux configurations de ce système sont donnés au tableau 5.1.

Tableau 5.1 : Coût des cartes du système mixte.

	Configuration Blazer	Configuration F5MS1
Carte Blazer	10,000$ US	-
Module XT10-4MCP	8,000$ CAN	8,000$ CAN
Module MDC40SS2	-	2 x 3,720$ US
Carte F5MS1 PCI	-	4,135$ US
Carte ZP10	8,000$	-
Total	30,000$ CAN	24,205$ CAN[2]

Notons que seuls les coûts du matériel sont pris en compte. Pour avoir une idée plus précise du coût réel, il faudrait rajouter le temps de conception du circuit qui doit être implanté sur les Xilinx, les outils de synthèse, etc. Les performances de ce système sont récapitulées dans le tableau 5.2.

---

[2] La valeur du dollar US est calculée comme étant 1.40 dollar canadien.

Tableau 5.2 : Performance du système mixte.

cycles/point	points/seconde
21	1,190,476

5.2 Système logiciel basé sur le TMS320C40

Le DSP TMS320C40 étant sur le marché depuis plusieurs années, une grande variété de produits sont disponibles. Par exemple la compagnie Data Translation a développé la carte Fulcrum, disposant d'un processeur TMS320C40, et de 1 Moctet de SRAM au prix de 2,000$ US. Cette carte peut être directement branchée sur un bus PCI, ce qui diminue la complexité de l'interface. Les performances de ce système sont récapitulées dans le tableau 5.3.

Tableau 5.3 : Performances obtenues avec le TMS320C40.

	code C (voir section 3.6)	code assembleur
Communications	70 cycles	44 cycles
CPU	94 cycles	46 cycles
Total	304,878 pts/sec.	555,555 pts/sec.

5.3 Système logiciel basé sur le ADSP-21060 (SHARC)

Au même titre que le TMS320C40, le SHARC profite d'une grande popularité, ce qui permet d'avoir une grande panoplie de produits sur le marché. Par exemple, la compagnie Bittware Research Systems, a mis sur le marché la carte "Blacktip" qui dispose d'un ADSP-21060.

Cette carte peut contenir jusqu'à 3.5 Moctets de SRAM, peut être accédée par un bus ISA, et coûte 1,270$ US.

Les performances de ce système sont récapitulées dans le tableau 5.4.

Tableau 5.4 : Performance du système basé sur un ADSP-21060

	Code C	Code Assembleur (voir section 4.3.2)
Communications	20 cycles	15 cycles
CPU	76 cycles	13 cycles
Total	416,666 pts/sec.	1,428,571 pts/sec.

5.4 Implantation sur FPGA

Plusieurs compagnies se sont lancées dans la fabrication de cartes contenant des FPGA qui peuvent être utilisées par l'usager. L'utilisateur n'est donc plus limité à un seul usage d'une carte, mais peut l'utiliser avec différentes configurations. Par exemple, on peut coupler cette carte avec un processeur et l'utiliser comme accélérateur, et on peut aussi l'utiliser pour faire du prototypage, etc. Malgré la vitesse relativement faible d'un FPGA, la versatilité d'une telle carte en fait un outil très efficace.

La compagnie Giga Operations a mis sur le marché la carte G900 qui dispose d'une interface PCI et sur laquelle il est possible d'installer des modules XMOD fabriqués par la même compagnie. Chacun de ces modules dispose de deux FPGA de la compagnie Xilinx (choix entre XC4010, XC4020 et XC4028), de 256 koctets de SRAM et de 8 Moctets de DRAM. L'ensemble de ces deux cartes avec l'option XC4020, coûte 3,000$ US. La performance de ce système est récapitulée au tableau 5.5.

Tableau 5.5 : Performance du système sur FPGA.

cycles/point	points/seconde
8	$2 \times 3.75 \times 10^6$

5.5 Comparaisons

Afin de déterminer lequel de ces systèmes est le plus intéressant, le tableau 5.6 montre le coût et la performance de chacun d'eux.

Tableau 5.6 : Comparaison coût-performance.

	Configuration mixte	Configuration TMS320C40	Configuration ADSP-21060	Configuration FPGA
coût	24,205$	2,000$ US	1,270$ US	3,000$ US
performance	1.19 Mpts./sec.	0.55 Mpts./sec.	1.42 Mpts./sec.	7.5 Mpts./sec.

D'après ce tableau, il est possible d'éliminer la solution logicielle avec le processeur TMS320C40, puisqu'elle n'atteint pas les performances visées. En effet, la performance obtenue par ce système est moins bonne que celle obtenue avec l'ADSP-21060, en grande partie due au fait que les unités arithmétiques du TMS320C40 fonctionnent avec une horloge de 25 MHz, tandis que celles du SHARC utilise une horloge de 40 MHz. De plus, le nombre de registres plus élevé dans ce dernier permet de diminuer considérablement le nombre d'instructions nécessaires à l'exécution de l'algorithme.

La deuxième remarque qu'il est possible de tirer du tableau 5.6, c'est que la solution mixte logicielle-matérielle coûte très cher par rapport aux autres solutions. En effet, la puissance de calcul disponible sur ce système nous laissait envisager de très bonnes performances. Dû

au nombre élevé de cycles reliés aux communications, la performance de ce système ne peut pas justifier son coût élevé.

Les performances du système basé seulement sur un FPGA sont très intéressantes. Pour un coût environ deux fois plus élevé que la solution logicielle avec SHARC, on obtient une performance plus de cinq fois meilleure. À première vue, ce système est le plus avantageux. En fait, le coût fournit au tableau 5.6 inclut seulement le coût du matériel. À ce montant, il faut rajouter les coûts reliés à la conception du circuit qui sera implanté dans le FPGA (outil de synthèse, temps de conceptions, etc.). Ces coûts ne sont pas négligeables, puisqu'ils seront certainement beaucoup plus élevés que le coût du matériel.

Si ce système est produit à une échelle suffisament grande, le coût non-récurrent dû à la conception du circuit devient négligeable, et le rapport coût-performance devient très intéressant.

De plus, le but à plus long terme de ce projet est la possibilité de transformer 10 millions de points par seconde. Afin d'obtenir cette performance, la seule modification à apporter à ce système c'est l'ajout d'un autre module XMOD au prix de 1,200 $ US, puisque la carte G900 supporte jusqu'à 16 modules XMOD. En fait, il serait possible d'atteindre avec deux modules XMOD, une performance de 15 Mpoints par seconde. Les produits de la compagnie Giga Operations n'ont pas été testés. Ces données sont basées sur les spécifications fournies par la compagnie.

Si au contraire le système est produit à quelques exemplaires seulement, les coûts de conception deviennent non négligeables. Une version qui est moins performante mais qui

peut être développée à des coûts moindres, devient plus intéressante. C'est le cas de la version logicielle avec ADSP-21060. Les coûts non-récurrents sont beaucoup moins élevés que la solution avec FPGA. Néanmoins, dans le cas où le débit désiré est de 10 millions de points par seconde, ce système ne peut pas être mis à jour. Une possibilité est d'utiliser un processeur SHARC monté sur une carte comportant plusieurs SHARCPAC. Cette solution est envisageable, mais beaucoup plus chère, puisque sept modules TIM possèdant un ADSP-21060 seraient nécessaires.

Chapitre 6 : Conclusion

La version de la caméra auto-synchronisée qui est présentement développée, par les chercheurs du CNRC, représente un défi de taille pour le système qui aura la tâche de transformer les données brutes qu'elle fournit, en des points de l'espace cartésien. Malgré la diminution de la complexité de l'algorithme de transformation original, ainsi que l'accélération du calcul des fonctions trigonométriques grâce aux LUT, le nombre d'opérations arithmétiques à effectuer sur chaque point reste grand. Ce mémoire a montré que l'objectif de 1 à 10 millions de transformations par seconde rend difficile l'utilisation d'implantations logicielles de haut niveau sur un processeur d'usage général. En effet, au chapitre 4, l'algorithme de transformation a été codé en langage assembleur et exécuté sur deux processeurs à virgule flottante à la fine pointe de la technologie, et un seul des deux a réussi à passer le cap des 1 million de points par seconde. Pour atteindre l'objectif de 10 millions de transformations par seconde en utilisant des processeurs d'usage général, il faut augmenter le nombre de processeurs. L'architecture mixte logicielle-matérielle étudiée dans ce mémoire utilise trois processeurs en parallèle. Malgré la puissance de calcul qu'offrait cette architecture, la performance obtenue atteint l'objectif de 1 million de points par seconde, mais elle est loin de ce que nous espérions. La communication entre les processeurs est le goulot d'étranglement de ce système. Il serait certainement possible d'obtenir de meilleures performances en développant une architecture qui utiliserait un système de communication inter-processeurs par mémoires partagées. Néanmoins, le bon fonctionnement d'un tel système dépend beaucoup de la performance du système d'arbitration qui donne accès à la

mémoire. Malgré la facilité d'implantation sur un processeur d'utilisation générale, il ne faut pas négliger l'approche matérielle qui a été développée au chapitre 4. En effet, il a été montré au chapitre 3 qu'une implantantation qui utilise des opérandes à virgule fixe de 20 bits, obtiendrait une précision suffisante. Ceci a pour effet de diminuer la taille des unités opératives, ce qui permet une implantation sur des FPGAs. Aucune étude n'a cependant été faite sur le type de FPGA qui fournirait les meilleures performances. Par exemple, la compagnie Altera fabrique maintenant des FPGAs qui disposent d'une quantité impressionante de mémoire interne dans lesquelles il serait possible de mettre les LUTs. De plus, le nombre de portes disponibles sur ce type de circuit augmente d'année en année. Il serait donc possible d'implanter plusieurs unités opératives en parallèle, qui chacune pourra transformer un point différent. Cette méthode a pour avantage de partager la mémoire dans laquelle se trouve les LUTs.

Dans une perspective purement basée sur les performances, le problème peut être résolu en utilisant un grand nombre d'unités opératives (circuit dédié, processeurs, etc.). Il ne faut cependant pas perdre de vue le coût d'implantation de ces systèmes. Le chapitre 5 de ce mémoire fait une étude très partielle du coût d'achat du matériel nécessaire à l'implantation des systèmes étudiés dans ce travail, mais ne tient pas compte du coût de développement (outils de synthèse, temps de développement, etc.). Une étude plus approfondie permettrait d'estimer plus précisément le coût réel de ces systèmes, et ainsi obtenir une idée plus précise du système qui a la meilleure relation coût-performance.

Bibliographie

ANALOG DEVICES (mars 1995). ADSP-2106x SHARC User's Manual.

ANALOG DEVICES (1997). Analog Products - ADDS-2106x SHARC EZ-KIT LITE. "http://analog.com/products".

BERALDIN J.-A, EL-HAKIM S. F. et COURNOYER L. (1993). Practical Range Camera Calibration. SPIE Proceeding, Videometrics, Vol. 2067, pp. 21-31.

BERALDIN J.-A, RIOUX M., BLAIS F. et COUVILLON R. A. (1994). Digital Three-Dimensional Imaging in the Infrared at the National Research Council of Canada. SPIE Proceeding, Vol. 2269, pp. 208-225.

BITTWARE (1997). Blacktip-PCI product. "http://www.bittware.com/products/sharc/btpci/ds1.htm".

BLINN J. (Nov. 1995). Three wrongs make a right. IEEE Computer Graphics and Applications. pp. 90-93.

CAVANAGH J. F. (1984). Digital Computer Arithmetic. McGraw-Hill inc., CA.

EMBEDDED SYSTEMS PROGRAMMING - ESP MAGAZINE (1997). DSP boards. "http://www.embedded.com/guides/dspbdsc.htm".

GIGA OPERATIONS (1997). "http://www.gigaops.com".

GOSLIN G. (1997). Using Xilinx FPGA's to design custom digital signal processing devices. Xilinx inc. "http://www.xilinx.com/appnotes/dspx5dev.htm".

HWANG K. (1993). Advanced Computer Architecture. McGraw-Hill inc., CA.

IMAGE & SIGNAL PROCESSING INC. (1994). Blazer - Reference Manual.

KHALI H. (Sept. 1995). Modélisation d'un système optique à base d'une caméra autosynchronisée. Rapport de stage au CNRC.

KHALI H., SAVARIA Y. et HOULE J.-L. A VLSI Chip for 3-D Camera Calibration.

LEVY M. (may 1997). DSP-Architecture Directory. EDN Magazine. Pp. 43-107.

LOUGHBOROUGH SOUND IMAGES. PC/DMCB2 product. "http://www.lsi-dsp.com/products".

MIROTECH MICROSYSTEMS INC. (1997). "http://www.mirotech.com/products".

RIOUX M. (Sept. 4-6 1990). Applications of Digital 3-D Imaging. Canadian Conf. on Electr. Comp. Eng., Ottawa, 37.3.1-37.3.10.

SPECTRUM SIGNAL PROCESSING, "http://www.spectrumsignal.com/sspcat".

TEXAS INSTRUMENTS (1991) TMS320C40 - User's guide.

TEXAS INSTRUMENTS (1997). Digital Signal Processing Solutions - TMS320C3x Tools. "http://www.ti.com/sc/docs/dsps/tools/c3x/c3xdsk.htm".

SYNOPSYS (1995). Logic Modeling - SmartModel Products - User's guide. Release 39.

XILINX (1996). Databook version 1.04.

Annexe A : Rappel théorique

Séries de Taylor et règle de Hörner

Les fonctions trigonométriques sont représentées par des séries de Taylor. Le nombre de facteurs de la série est déterminé par la précision désirée, ce qui implique un plus grand nombre d'opérations pour une plus grande précision.

$$\sin(x) = \sum_{k=0}^{\infty} A_k \, x^{(2k+1)} \; avec \; A_k = \frac{(-1)^k}{(2k+1)!}$$
$$\cos(x) = \sum_{k=0}^{\infty} B_k \, x^{(2k)} \; avec \; B_k = \frac{(-1)^k}{(2k)!}$$

Les développements en série de Taylor des 7 premiers termes des fonctions sin(x) et cos(x) donnent donc :

$$\sin(x) = x + A_1 x^3 + A_2 x^5 + A_3 x^7 + A_4 x^9 + A_5 x^{11} + A_6 x^{13}$$
$$\cos(x) = 1 + B_1 x^2 + B_2 x^4 + B_3 x^6 + B_4 x^8 + B_5 x^{10} + B_6 x^{12}$$

La règle de Hörner est en fait une factorisation qui permet de diminuer le nombre d'opérations à effectuer. En utilisant cette règle pour les séries précédentes, on voit que le nombre d'additions reste à 6 tandis que le nombre de multiplications passe de 18 à 8 dans le cas de la fonction *sinus* et de 17 à 7 dans le cas du cosinus :

$$\sin(x) = ((((((A_6 x^2 + A_5)x^2 + A_4)x^2 + A_3)x^2 + A_2)x^2 + A_1)x^2 + 1)x$$
$$\cos(x) = (((((B_6 x^2 + B_5)x^2 + B_4)x^2 + B_3)x^2 + B_2)x^2 + B_1)x^2 + 1$$

Il est à noter que les sept premiers termes de ces séries offrent de très bon résultats pour des x compris dans l'intervalle $[-3\pi/2, 3\pi/2]$, mais hors de ces bornes les résultats divergent très rapidement.

Algorithme de division Newton-Raphson[1]

Afin d'obtenir le quotient A/B, la méthode par itération Newton-Raphson propose de calculer A(1/B). Cette méthode est très pratique pour améliorer une première approximation d'une

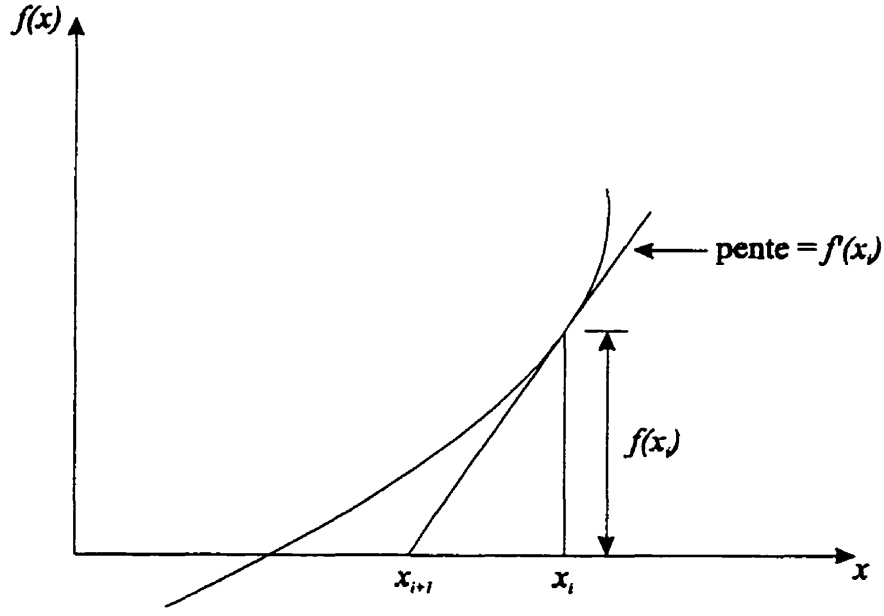


Figure A.1 : Méthode Newton-Raphson.

équation $f(x) = 0$.

Le point x_i de la figure A.1, est la première approximation de la racine de la fonction $f(x)$. Si on trace la droite tangente à la courbe au point $x = x_i$, alors la tangente coupera l'axe des x

[1](Cavanagh, 1984) p. 284

au point x_{i+1} . Ce point est une meilleure approximation de la racine $f(x) = 0$ que le point x_i.

La pente de la tangente est donnée par :

$$f'(x_i) = \frac{f(x_i)}{x_i - x_{i+1}} \tag{1}$$

ou bien

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} \tag{2}$$

Nous pouvons définir une fonction $f(x)$ qui permettra d'obtenir le quotient 1/B.

$$f(x) = \frac{1}{x} - B \tag{3}$$

La racine de cette fonction lorsque f(x) = 0 est :

$$x = \frac{1}{B}$$

ce qui correspond au résultat recherché. Afin d'utiliser la méthode Newton-Raphson, il faut obtenir la dérivé de la fonction (3).

$$\begin{aligned} f'(x) &= f'(x^{-1}) - f'(B) \\ &= (-1)(x^{-2}) - 0 \\ &= \frac{-1}{x^2} \end{aligned} \tag{4}$$

En substituant les équations (3) et (4) dans l'équation (2), on obtient :

$$x_{i+1} = x_i - \left[\frac{(1/x_i) - B}{-(1/x^2)}\right]$$
$$= x_i\,(2 - Bx_i) \qquad (5)$$

La première approximation est cruciale pour cette méthode. Pour que le processus converge, la première approximation x_0 doit être comprise dans l'intervalle $0 < x_0 < 2/B$.

Le dénominateur B doit être une fraction positive et normalisée afin d'être dans l'intervalle

$$1/2 \leq B \leq 1$$

Donc, l'inverse de B sera borné par

$$1 < 1/B \leq 2$$

La valeur initiale x_0 doit être raisonnablement précise afin que le nombre d'itérations ne soit pas trop grand. Pour diminuer l'erreur initiale ainsi que le temps de calcul, x_0 peut être mis dans une LUT. Le tableau A.1 montre un exemple de LUT.

Tableau A.1 : Table partielle pour l'approximation de 1/B

entrée B	approximation de 1/B
0.1000	1.1111
0.1001	1.1100
0.1010	1.1001
0.1011	1.0111
0.1100	1.0101
0.1101	1.0011
0.1110	1.0010
0.1111	1.0001

Puisque

$$\lim_{i \to \infty} x_i = \frac{1}{B}$$

il faut déterminer une méthode pour évaluer le moment où devra s'arrêter le processus. Si ε est une valeur très petite qui peut représenter l'erreur d'approximation, le processus s'arrêtera lorsque

$$|1 - Bx_n| \leq \varepsilon$$

où x_n est l'approximation finale de l'inverse B.

Annexe B : Algorithme Direct codé en langage C afin d'être exécuté sur le DSP TMS320C40

```
#define beta 1
#define gamma 1
#define T  1
#define S  1
#define Hx 1
#define Hy 1
#define Hz 1

unsigned int i, j, p, l;
double X, Y, Z;
double X_inf, Xo, Z_inf, Zo, phi, theta, Zs;
double t1, t2, t3, t4, t5, t6, t7, t8, t9, t10, t11, t12, t13;
double t14, t15, t16;
double si(double);
double co(double);

double A1 = -0.1666;
double A2 = 8.33E-3;
double A3 = -1.9841E-4;
double A4 = 2.7557E-6;
double A5 = -2.5052E-8;
double A6 = 1.6059E-10;

double B1 = -0.5;
double B2 = 0.04166;
double B3 = -1.3888E-3;
double B4 = 2.4801E-5;
double B5 = -2.7557E-7;
double B6 = 2.0877E-9;

void main(void)
{
    double p_inf = 33871.9;
    double phi_t[3] = {0.0845076, -0.000459854, 1.68086E-11};
    double theta_t[4] = {0.113598, -0.000378664, -3.27184E-11,
7.7394E-7};

volatile int   *port1_sts = (int *)0x100050;
    volatile float *port1_in  = (void *)0x100051;
    volatile float *port1_out = (void *)0x100052;
    volatile int   *port4_sts = (int *)0x100080;
    volatile int   *port4_in  = (int *)0x100081;
    volatile int   *port4_out = (int *)0x100082;

    t3 = beta -gamma;
    t5 = gamma / 2;
    t7 = S*co(beta);
    t8 = 1/co(t3);
```

```
      t9 = T/si(gamma);
      t14 = si(t3);
      t15 = co(gamma);
      t16 = co(t5);

      for(l = 0;l < 24; l++)
      {
        /* Ouvre le port de  communication */
        *port4_sts = (*port4_sts & 0xfff7);

        /* attendre qu'une donnee soit arrivee*/
        while(!(*port4_sts & 0x1E00));
              /* va lire p dans le port #4*/
              p = *port4_in;

        /* attendre qu'une donnee soit arrivee*/
        while(!(*port4_sts & 0x1E00));
              /* va lire j dans le port #4*/
              j = *port4_in;

        /* attendre qu'une donnee soit arrivee*/
        while(!(*port4_sts & 0x1E00))
              /* va lire i dans le port #4*/
              i = *port4_in;

        theta = theta_t[0] + i * (theta_t[1] + (i*i*theta_t[2])) +
(p*p*p*theta_t[3]);
        phi = phi_t[0] + j * (phi_t[1] + (phi_t[2] * j * j));

        t1 = 2* theta;
        t2 = beta - t1;
        t4 = 2 * phi;
        t6 = t5 - theta;
        t10 = 1 / (p_inf - p);
        t11 = co(t4);
        t12 = si(t4);

        X_inf = ((T*co(t2)) + (t7*si(t6))) * t8;
        Z_inf = ((t7 * co(t6)) - (T * si(t2) * t14)) * t8;
        Xo = si(t1) * t9;
        Zo = (co(t1) + t15) * -t9;
        Zs = Z_inf + ((p_inf * t10) * (Zo - Z_inf));
        X = X_inf + ((p_inf * t10) * (Xo - X_inf));
        t13 = Zs - (Hx * co(t5)) - Hy;
        Y = (t13 * t12) - (Hz * t11) + Hy;
        Z = (t13 * t11) - (Hz *t12) + Hz;
        *port1_out = X;
        *port1_out = Y;
        *port1_out = Z;
      }
}
```

```
double si(double x)
{
      double x2 = x * x;
      double sin = ((((((A6*x2+A5)*x2+A4)*x2+A3)*x2+A2)*x2+A1)*x2+1)*x;
      return sin;
}

double co(double x)
{
      double x2 = x * x;
      double cos = ((((((B6*x2+B5)*x2+B4)*x2+B3)*x2+B2)*x2+B1)*x2)+1;
      return cos;
}
```

Annexe C : Algorithme FIT codé en langage C pour être exécuté sur le DSP TMS320C40

```
unsigned int i, j, p, l;
double G, X, T, Y, Z;
double Xg, Xo, Zg, Zo, Hy, Hz, phi_t;
double si(double);
double co(double);

double A1 = -0.1666;
double A2 = 8.33E-3;
double A3 = -1.9841E-4;
double A4 = 2.7557E-6;
double A5 = -2.5052E-8;
double A6 = 1.6059E-10;

double B1 = -0.5;
double B2 = 0.04166;
double B3 = -1.3888E-3;
double B4 = 2.4801E-5;
double B5 = -2.7557E-7;
double B6 = 2.0877E-9;


void main(void)
{
double p_inf = 33871.9;
double phi[3] = {0.0845076, -0.000459854, 1.68086E-11};
double A[6] = {0.899128E+6, -0.613627E+4, -0.235729, 0.894595E-4,
-0.464256E-6, 0.105075E-9};
double B[6] = {0.178122E+1, -0.0170118, -0.974965E-5, -0.103779E-8,
0.131898E-10, 0.235288E-14};
double C[6] = {-0.162615E+8, -0.632839E+3, 0.225903E+1, -0.00117959,
0.290291E-6, 0.835121E-10};
double D[6] = {-0.476869E+2, -0.0253391, 0.837623E-5, 0.413716E-7,
0.951576E-12, -0.196399E-14};
double E[6] = {0.136153E+3, 0.0166062, 0.706491E-6, -0.119281E-8,
-0.625863E-13, 0.695164E-16};
double F[6] = {-0.33112E+2, 0.000235217, 0.388218E-5, -0.133168E-10,
-0.365501E-12, 0.242089E-15};

    volatile int   *port1_sts = (int *)0x100050;
    volatile float *port1_in  = (void *)0x100051;
    volatile float *port1_out = (void *)0x100052;
    volatile int   *port4_sts = (int *)0x100080;
    volatile int   *port4_in  = (int *)0x100081;
    volatile int   *port4_out = (int *)0x100082;

    for(l = 0;l < 24; l++)
```

```
    {
      /* ouvre le port de communication */
      *port4_sts = (*port4_sts & 0xfff7);

      /* attendre qu'une donnee soit arrivee*/
      while(!(*port4_sts & 0x1E00));
            /* va lire p dans le port #4*/
            p = *port4_in;

      /* attendre qu'une donnee soit arrivee*/
      while(!(*port4_sts & 0x1E00));
            /* va lire j dans le port #4*/
            j = *port4_in;

      /* attendre qu'une donnee soit arrivee*/
      while(!(*port4_sts & 0x1E00));
            /* va lire i dans le port #4*/
            i = *port4_in;

      phi_t = phi[0] + j * (phi[1] + (phi[2] * j * j));

      Xg = (((((((((A[5] * i) + A[4]) * i) + A[3]) * i) + A[2]) * i) +
A[1]) * i) + A[0];
      Xo = (((((((((B[5] * i) + B[4]) * i) + B[3]) * i) + B[2]) * i) +
B[1]) * i) + B[0];
      Zg = (((((((((C[5] * i) + C[4]) * i) + C[3]) * i) + C[2]) * i) +
C[1]) * i) + C[0];
      Zo = (((((((((D[5] * i) + D[4]) * i) + D[3]) * i) + D[2]) * i) +
D[1]) * i) + D[0];
      Hy = (((((((((E[5] * j) + E[4]) * j) + E[3]) * j) + E[2]) * j) +
E[1]) * j) + E[0];
      Hz = (((((((((F[5] * j) + F[4]) * j) + F[3]) * j) + F[2]) * j) +
F[1]) * j) + F[0];

        G = 1 / (p - p_inf);
        X = Xg * G - Xo;
        T = (G * Zg) - Zo;
        Y = T * si(phi_t) + Hy;
        Z = T * co(phi_t) + Hz;
        *port1_out = X;
        *port1_out = Y;
        *port1_out = Z;
    }
}

double si(double x)
{
      double x2 = x * x;
      double sin = (((((A5*x2+A4)*x2+A3)*x2+A2)*x2+A1)*x2+1)*x;
      return sin;
}
```

```
double co(double x)
{
      double x2 = x * x;
      double cos = (((((B5*x2+B4)*x2+B3)*x2+B2)*x2+B1)*x2)+1;
      return cos;
}
```

Annexe D : Calcul de précision

De la même façon qu'on a pu obtenir Xg à la section 3.7, il est possible d'obtenir la mise à l'échelle des autres développements en série de l'algorithme Fit. Ces résultats ne sont valables que lorsque la caméra est calibrée pour un objet d'un mètre cube à une distance d'un mètre (voir tableau 3.4). Voici un résumé des résultats obtenus pour ces séries :

$$X_g(i) = (((((A_5 i \times 2^{-12} + A_4) i \times 2^{-9} + A_3) i \times 2^{-11} + A_2) i \times 2^{-14} + A_1) i \times 2^{-10} + A_0) \times 2^5$$
$$X_0(i) = (((((B_5 i \times 2^{-13} + B_4) i \times 2^{-8} + B_3) i \times 2^{-11} + B_2) i \times 2^{-11} + B_1) i \times 2^{-11} + B_0) \times 2^{-12}$$
$$Z_g(i) = (((((C_5 i \times 2^{-14} + C_4) i \times 2^{-14} + C_3) i \times 2^{-15} + C_2) i \times 2^{-10} + C_1) i \times 2^{-2} + C_0) \times 2^7$$
$$Z_0(i) = (((((D_5 i \times 2^{-9} + D_4) i \times 2^{-15} + D_3) i \times 2^{-9} + D_2) i \times 2^{-10} + D_1) i \times 2^{-15} + D_0) \times 2^{-8}$$
$$H_y(j) = (((((E_5 j \times 2^{-10} + E_4) j \times 2^{-14} + E_3) j \times 2^{-9} + E_2) j \times 2^{-15} + E_1) j \times 2^{-13} + E_0) \times 2^{-10}$$
$$H_z(j) = (((((F_5 j \times 2^{-10} + F_4) j \times 2^{-9} + F_3) j \times 2^{-15} + F_2) j \times 2^{-10} + F_1) j \times 2^{-25} + F_0) \times 2^{-17}$$

	A	*B*	*C*	*D*	*E*	*F*
5	236,607 × 2^{-51}	173,611 × 2^{-66}	24,069 × 2^{-48}	-144,916 × 2^{-66}	164,140 × 2^{-71}	142,904 × 2^{-69}
4	-255,227 × 2^{-39}	118,803 × 2^{-53}	4,987 × 2^{-34}	137,136 × 2^{-57}	-144,314 × 2^{-61}	-210,696 × 2^{-59}
3	93,056 × 2^{-30}	-36,513 × 2^{-45}	-1,236 × 2^{-20}	181,954 × 2^{-42}	-167,873 × 2^{-47}	-14,993 × 2^{-50}
2	-123,589 × 2^{-19}	-167,497 × 2^{-34}	72 × 2^{-5}	71,951 × 2^{-33}	194,192 × 2^{-38}	133,390 × 2^{-35}
1	-196,456 × 2^{-5}	-142,705 × 2^{-23}	-19 × 2^5	-212,559 × 2^{-23}	139,302 × 2^{-23}	7,892 × 2^{-25}
0	28,097 × 2^5	3,647 × 2^{-12}	-127,042 × 2^7	-12,207 × 2^{-8}	139,420 × 2^{-10}	-8,476 × 2^{-8}

$$\phi(j) = \phi_0 + \phi_1 j + \phi_3 j^3 = \phi_0 + j(\phi_1 + \phi_3 j^2)$$
$$\sin(\phi(j)) = ((((((G_5 \phi^2 + G_4)\phi^2 + G_3)\phi^2 + G_2)\phi^2 + G_1)\phi^2 + G_0)\phi^2 + 1)\phi$$
$$\cos(\phi(j)) = (((((H_5 \phi^2 + H_4)\phi^2 + H_3)\phi^2 + H_2)\phi^2 + H_1)\phi^2 + H_0)\phi^2 + 1$$

	ϕ	G	H
5	-	$-242{,}121 \times 2^{-42}$	$-151{,}496 \times 2^{-39}$
4	-	$189{,}370 \times 2^{-36}$	$213{,}038 \times 2^{-33}$
3	$151{,}398 \times 2^{-53}$	$-213{,}041 \times 2^{-30}$	$-186{,}401 \times 2^{-27}$
2	-	$139{,}754 \times 2^{-24}$	$174{,}734 \times 2^{-22}$
1	$-246{,}882 \times 2^{-29}$	$-174{,}762 \times 2^{-20}$	$-131{,}072 \times 2^{-18}$
0	$88{,}612 \times 2^{-20}$	$16{,}384 \times 2^{-14}$	$16{,}384 \times 2^{-14}$

Annexe E : Programme VHDL non-synthétisable pour déterminer la perte de précision due à l'utilisation d'un format à virgule fixe

Description du programme VHDL pour déterminer la précision

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

entity fit is
  port(
       enable : in std_logic;
       i : in std_logic_vector(8 downto 0);
       j : in std_logic_vector(8 downto 0);
       p : in std_logic_vector(8 downto 0)
       );
end fit;

architecture rtl of fit is

  subtype coef is std_logic_vector(23 downto 0);
  type mat is array  (5 downto 0) of coef;
  type angle is array (2 downto 0) of coef;
  type approx is array (4 downto 0) of coef;

  FUNCTION  add(M:std_logic_vector(23 downto 0); N:std_logic_vector(23
downto 0)) RETURN std_logic_vector is
    variable result : std_logic_vector(23 downto 0);
  begin

    if (M(23) = '1') then
      if (N(23) = '1') then
      result(22 downto 0) := M(22 downto 0) + N(22 downto 0);
      result(23) := '1';
      else
      if(M(22 downto 0) > N(22 downto 0)) then
        result(22 downto 0) := M(22 downto 0) - N(22 downto 0);
        result(23) := '1';
      else
        result(22 downto 0) := N(22 downto 0) - M(22 downto 0);
        result(23) := '0';
      end if;
      end if;
    else
      if (N(23) = '0') then
      result(22 downto 0) := M(22 downto 0) + N(22 downto 0);
      result(23) := '0';
```

```
      else
      if(M(22 downto 0) > N(22 downto 0)) then
        result(22 downto 0) := M(22 downto 0) - N(22 downto 0);
        result(23) := '0';
      else
        result(22 downto 0) := N(22 downto 0) - M(22 downto 0);
        result(23) := '1';
      end if;
      end if;
    end if;
    return result;

  end add;

  FUNCTION  sub(M:std_logic_vector(23 downto 0); N:std_logic_vector(23
downto 0)) RETURN std_logic_vector is
    variable result : std_logic_vector(23 downto 0);
  begin

    if (M(23) = '0') then
      if (N(23) = '1') then
      result(22 downto 0) := M(22 downto 0) + N(22 downto 0);
      result(23) := '0';
      else
      if(M(22 downto 0) > N(22 downto 0)) then
        result(22 downto 0) := M(22 downto 0) - N(22 downto 0);
        result(23) := '0';
      else
        result(22 downto 0) := N(22 downto 0) - M(22 downto 0);
        result(23) := '1';
      end if;
      end if;
    else
      if (N(23) = '0') then
      result(22 downto 0) := M(22 downto 0) + N(22 downto 0);
      result(23) := '1';
      else
      if(M(22 downto 0) > N(22 downto 0)) then
        result(22 downto 0) := M(22 downto 0) - N(22 downto 0);
        result(23) := '1';
      else
        result(22 downto 0) := N(22 downto 0) - M(22 downto 0);
        result(23) := '0';
      end if;
      end if;
    end if;
    return result;

  end sub;

  FUNCTION  mul(M:std_logic_vector(23 downto 0); N:std_logic_vector(8
downto 0);cnt:std_logic_vector(4 downto 0)) RETURN std_logic_vector is
```

```
    variable tmp, result_tmp : std_logic_vector(32 downto 0);
    variable result : std_logic_vector(23 downto 0);
  begin

    tmp := (others => '0');
    tmp(31 downto 0) := M(22 downto 0) * N;
    result_tmp := SHR(tmp,cnt);
    result := result_tmp(23 downto 0);
    result(23) := M(23);
    return result;

  end mul;

  FUNCTION  mul48(M:std_logic_vector(23 downto 0); N:std_logic_vector(23
downto 0);cnt:std_logic_vector(5 downto 0))
    RETURN std_logic_vector is

    variable tmp, result_tmp : std_logic_vector(45 downto 0);
    variable result : std_logic_vector(23 downto 0);
  begin

    tmp := (others => '0');
    result_tmp := (others => '0');
    tmp(45 downto 0) := M(22 downto 0) * N(22 downto 0);
    result_tmp := SHR(tmp,cnt);
    result := result_tmp(28 downto 5);
    result(23) := M(23) xor N(23);
    return result;

  end mul48;


  FUNCTION  j2(M:std_logic_vector(8 downto 0)) RETURN std_logic_vector
is

    variable tmp, result_tmp : std_logic_vector(17 downto 0);
    variable result : std_logic_vector(23 downto 0);
  begin

    tmp := (others => '0');
    result := (others => '0');
    tmp := M * M;
    result(22 downto 5) := tmp;
    return result;

  end j2;


begin
  enable_proc :  process(enable)

    variable A,B,C,D,E,F : mat;
```

```
    variable ph : angle;
    variable K, L : approx;
    variable Xg1, Xg2, Xg3, Xg4, Xg5, Xg6, Xg7, Xg8, Xg9, Xg :
std_logic_vector(23 downto 0);
    variable Xo1, Xo2, Xo3, Xo4, Xo5, Xo6, Xo7, Xo8, Xo9, Xo :
std_logic_vector(23 downto 0);
    variable Zg1, Zg2, Zg3, Zg4, Zg5, Zg6, Zg7, Zg8, Zg9, Zg :
std_logic_vector(23 downto 0);
    variable Zo1, Zo2, Zo3, Zo4, Zo5, Zo6, Zo7, Zo8, Zo9, Zo :
std_logic_vector(23 downto 0);
    variable Hy1, Hy2, Hy3, Hy4, Hy5, Hy6, Hy7, Hy8, Hy9, Hy :
std_logic_vector(23 downto 0);
    variable Hz1, Hz2, Hz3, Hz4, Hz5, Hz6, Hz7, Hz8, Hz9, Hz :
std_logic_vector(23 downto 0);
    variable phi1, phi2, phi3, phi4, phi : std_logic_vector(23 downto
0);
    variable si1, si2, si3, si4, si5, si6, si7, si8, si9, si10, si11,
si : std_logic_vector(23 downto 0);
    variable co1, co2, co3, co4, co5, co6, co7, co8, co9, co :
std_logic_vector(23 downto 0);
    variable X1, X2, X, G1, G, Y1, Y2, Y, Z1, Z2, Z :
std_logic_vector(23 downto 0);
    variable inv : std_logic_vector(23 downto 0);
  begin

      A := ("011100111000011111111011","111111001001111101101101",
            "001011101110011100001101","101111000101100010111100",
            "110111111110110100010100","000011011011100000111000");

      B := ("010101001100010101111100","001110100000001001100100",
            "100100011101010000111111","110100011100100100110110",
            "110001011010111000101010","000000111000111111111100");

      C := ("000010111100000010110000", "000000100110111101100101",
            "100000001001101010011100", "000000000000100100001001",
            "100000000000001001111000", "101111100000100001011111");

      D := ("110001101100001010011100", "010000101111011000010001",
            "010110001101100001000111", "001000110010000111101000",
            "111001111100100111111000", "100001011111101011111011");

      E := ("010100000100101100111110", "110001100111011101000101",
            "110100011111100000100010", "010111101101001000011000",
            "010001000000010011011100", "010001000001001110010101");

      F := ("010001011100011100000011", "111001101110000100011111",
            "100001110101001000101100", "010000010010000111010110",
            "000000111101101010010010", "100001000010001110010101");

      K := ("111101100011100100101000", "010111000111011101001000",
            "111010000000011000100011", "010001000011110101000110",
            "110101010101010101010101");
```

```
L := ("110010011111100100000110", "011010000000010111011110",
      "110110110000010000110010", "010101010101000111010110",
      "110000000000000000000000");

ph := ("010010011110110011001101", "111110001000110001000111",
       "001010110100010010010100");

if (enable = '1') then
-- Calcul de Xg (Multiplier Xg par 2^5 pour obtenir resultat)
Xg1 := mul(A(5),i,"01100");
Xg2 := add(Xg1,A(4));
Xg3 := mul(Xg2,i,"01001");
Xg4 := add(Xg3,A(3));
Xg5 := mul(Xg4,i,"01011");
Xg6 := add(Xg5,A(2));
Xg7 := mul(Xg6,i,"01110");
Xg8 := add(Xg7,A(1));
Xg9 := mul(Xg8,i,"01010");
Xg  := add(Xg9,A(0));

-- Calcul de Xo (multiplier Xg par 2^-12 pour obtenir la reponse)
Xo1 := mul(B(5),i,"01101");
Xo2 := add(Xo1,B(4));
Xo3 := mul(Xo2,i,"01000");
Xo4 := add(Xo3,B(3));
Xo5 := mul(Xo4,i,"01011");
Xo6 := add(Xo5,B(2));
Xo7 := mul(Xo6,i,"01011");
Xo8 := add(Xo7,B(1));
Xo9 := mul(Xo8,i,"01011");
Xo  := add(Xo9,B(0));

-- Calcul de Zg (Multiplier Zg par 2^7 pour obtenir resultat)
Zg1 := mul(C(5),i,"01110");
Zg2 := add(Zg1,C(4));
Zg3 := mul(Zg2,i,"01110");
Zg4 := add(Zg3,C(3));
Zg5 := mul(Zg4,i,"01111");
Zg6 := add(Zg5,C(2));
Zg7 := mul(Zg6,i,"01010");
Zg8 := add(Zg7,C(1));
Zg9 := mul(Zg8,i,"00010");
Zg  := add(Zg9,C(0));

-- Calcul de Zo (Multiplier Zo par 2^-9 pour obtenir resultat)
Zo1 := mul(D(5),i,"01001");
Zo2 := add(Zo1,D(4));
Zo3 := mul(Zo2,i,"01111");
Zo4 := add(Zo3,D(3));
Zo5 := mul(Zo4,i,"01001");
Zo6 := add(Zo5,D(2));
Zo7 := mul(Zo6,i,"01010");
```

```
Zo8 := add(Zo7,D(1));
Zo9 := mul(Zo8,i,"01111");
Zo  := add(Zo9,D(0));

-- Calcul de Hy (multiplier Hy par 2^-10 pour obtenir la reponse)
Hy1 := mul(E(5),j,"01010");
Hy2 := add(Hy1,E(4));
Hy3 := mul(Hy2,j,"01110");
Hy4 := add(Hy3,E(3));
Hy5 := mul(Hy4,j,"01001");
Hy6 := add(Hy5,E(2));
Hy7 := mul(Hy6,j,"01111");
Hy8 := add(Hy7,E(1));
Hy9 := mul(Hy8,j,"01101");
Hy  := add(Hy9,E(0));

-- Calcul de Hz (multiplier Hz par 2^-8 pour obtenir la reponse)
Hz1 := mul(F(5),j,"01010");
Hz2 := add(Hz1,F(4));
Hz3 := mul(Hz2,j,"01001");
Hz4 := add(Hz3,F(3));
Hz5 := mul(Hz4,j,"01111");
Hz6 := add(Hz5,F(2));
Hz7 := mul(Hz6,j,"01010");
Hz8 := add(Hz7,F(1));
Hz9 := mul(Hz8,j,"10001");
Hz  := add(Hz9,F(0));


-- Calcul de phi (multiplier le resultat par 2^-20 pour obtenir la
-- reponse)
phi1 := j2(j);
phi2 := mul48(phi1,ph(2),"011000");
phi3 := add(phi2,ph(1));
phi4 := mul(phi3,j,"01001");
phi := add(phi4,ph(0));

-- Calcul de sinus (multiplier le resultat par 2^-20 pour obtenir
-- la reponse)
si1 := mul48(phi,phi,"010001");
si2 := mul48(K(4),si1,"011101");
si3 := add(K(3),si2);
si4 := mul48(si3,si1,"011101");
si5 := add(K(2),si4);
si6 := mul48(si5,si1,"011101");
si7 := add(K(1),si6);
si8 := mul48(si7,si1,"011011");
si9 := add(K(0),si8);
si10 := mul48(si9,si1,"011101");
si11 := add(si10,"000010000000000000000000");
si := mul48(si11,phi,"001110");
```

```
        -- Calcul de cosinus (multiplier le resultat par 2^-14 pour
        -- obtenir la reponse)
        co1 := mul48(L(4),si1,"011001");
        co2 := add(L(3),co1);
        co3 := mul48(co2,si1,"011101");
        co4 := add(L(2),co3);
        co5 := mul48(co4,si1,"011100");
        co6 := add(L(1),co5);
        co7 := mul48(co6,si1,"011011");
        co8 := add(L(0),co7);
        co9 := mul48(co8,si1,"011011");
        co := add(co9,"000010000000000000000000");


        -- Calcul de 1/(p - P_inf) (multiplier le resultat par 2^-33 pour
        -- obtenir la reponse)
        if (p = "111111111") then
          -- Si p = 511 ; inv = -2.9975e-5
          inv := "111111011011100101101001";
        else
          -- Si p = 0 ; inv =-2.9523e-5
          inv := "111110111101010000010100";
        end if;

        -- Calcul de X (multiplier X2 par 2^-12 pour obtenir la reponse)
        X1 := mul48(Xg,inv,"010000");
        X2 := sub(X1,Xo);
        X(22 downto 0) := SHR(X2(22 downto 0),"1100");
        X(23) := X2(23);

        -- Calcul de (Zg/(p - P_inf)) - Zo (multiplier G par 2^-8 pour
        -- obtenir la reponse)
        G1 := mul48(Zg,inv,"010010");
        G := sub(G1,Zo);

        -- Calcul de Y (multiplier Y2 par 2^-10 pour obtenir la reponse)
        Y1 := mul48(G,si,"010010");
        Y2 := add(Y1,Hy);
        Y(22 downto 0) := SHR(Y2(22 downto 0),"1010");
        Y(23) := Y2(23);

        -- Calcul de Z (multiplier Z2 par 2^-8 pour obtenir la reponse)
        Z1 := mul48(G,co,"001110");
        Z2 := add(Z1,Hz);
        Z(22 downto 0) := SHR(Z2(22 downto 0),"1000");
        Z(23) := Z2(23);
        end if;

    end process;

end rtl;
```

Annexe F : Assembleur généré à partir du langage C pour l'algorithme FIT avec LUT

```
*******************************************************
*    TMS320C40 C COMPILER     Version 4.60
*******************************************************
;     C:\LOGICIEL\DSPTOOLS\ac30.exe -v40 -x proc2.c
C:\WINDOWS\TEMP\proc2.if
;     C:\LOGICIEL\DSPTOOLS\opt30.exe -v40 -s -o3
C:\WINDOWS\TEMP\proc2.if C:\WINDOWS\TEMP\proc2.opt
;     C:\LOGICIEL\DSPTOOLS\cg30.exe -v40 -n C:\WINDOWS\TEMP\proc2.opt
proc2.asm C:\WINDOWS\TEMP\proc2.tmp
      .version    40
FP    .set        AR3
      .globl      _p_inf
      .globl      _Xg
      .globl      _Xo
      .globl      _Zg
      .globl      _Zo
      .globl      _Hy
      .globl      _Hz
      .globl      _si
      .globl      _co
      .globl      _const1
      .globl      _const2
      .globl      _i
      .globl      _j
      .globl      _p
      .globl      _l
      .globl      _p_prime
      .globl      _G
      .globl      _X
      .globl      _T
      .globl      _Y
      .globl      _Z
      .globl      _main
      .globl      _G
      .bss  _G,1
      .globl      _T
      .bss  _T,1
      .globl      _X
      .bss  _X,1
      .globl      _Y
      .bss  _Y,1
      .globl      _Z
      .bss  _Z,1
      .globl      _i
      .bss  _i,1
      .globl      _j
      .bss  _j,1
      .globl      _l
```

```
        .bss  _l,1
        .globl      _p
        .bss  _p,1
        .globl      _p_prime
        .bss  _p_prime,1
        .globl      _main
*******************************************************
* FUNCTION DEF : _main
*******************************************************
_main:
        PUSH  AR4
        PUSH  AR5
        PUSH  AR6
        PUSH  AR7
*
* AR2 assigned to variable  port4_sts
* AR4 assigned to variable  port4_in
* AR5 assigned to variable  port1_out
* AR6 assigned to temp var  K$69
* AR7 assigned to temp var  K$64
* IR1 assigned to temp var  C$1
* IR1 assigned to temp var  C$2
* IR1 assigned to temp var  C$3
* IR1 assigned to temp var  Y$0
* R9  assigned to temp var  C$4
* R9  assigned to temp var  C$5
* R9  assigned to temp var  C$6
* R9  assigned to temp var  Y$1
* R9  assigned to temp var  Y$2
* R9  assigned to temp var  Y$3
* R9  assigned to temp var  Y$4
* R9  assigned to temp var  Y$5
*
*** 16      ----------------------    l = 0u;
        STIK  0,@_l
***     ----------------------    K$64 = &co[0];
        LDA   @CONST+0,AR7
***     ----------------------    K$69 = &Hz[0];
        LDA   @CONST+1,AR6
*** 11      ----------------------    port1_out = (float*)1048658;
        LDA   @CONST+2,AR5
*** 12      ----------------------    port4_sts = (int*)1048704;
        LDA   @CONST+3,AR2
*** 13      ----------------------    port4_in = (int*)1048705;
        LDA   @CONST+4,AR4
L2:
***     ----------------------g2:
*** 18      ----------------------    *port4_sts = *port4_sts&0xfff7;
        LDI   @CONST+5,R0
        AND   R0,*AR2,R1
        STI   R1,*AR2
L4:
```

```
***   ----------------------g4:
*** 19      ----------------------    if ( !(*port4_sts&0x1e00) ) goto
g4;
      LDI   7680,R0
      TSTB  R0,*AR2
      BZ    L4
*** 20      ----------------------    p = *port4_in;
      LDI   *AR4,R1
      STI   R1,@_p
L7:
***   ----------------------g7:
*** 22      ----------------------    if ( !(*port4_sts&0x1e00) ) goto
g7;
      LDI   7680,R0
      TSTB  R0,*AR2
      BZ    L7
*** 23      ----------------------    j = *port4_in;
      LDI   *AR4,R1
      STI   R1,@_j
L10:
***   ----------------------g10:
*** 25      ----------------------    if ( !(*port4_sts&0x1e00) ) goto
g10;
      LDI   7680,R0
      TSTB  R0,*AR2
      BZ    L10
*** 26      ----------------------    i = Y$0 = *port4_in;
      LDA   *AR4,IR1
      STI   IR1,@_i
*** 28      ----------------------    C$6 = (float)Y$0;
      FLOAT IR1,R9
      LDFLT @CONST+6,R1
      LDFGE 0,R1
      ADDF  R1,R9
*** 28      ----------------------    p_prime = C$5 =
(float)p+C$6*(const1+C$6*C$6*const2);
      FLOAT @_p,R1
      LDFLT @CONST+6,R2
      LDFGE 0,R2
      ADDF  R2,R1
      MPYF  R9,R9,R2
      MPYF  @_const2,R2
      ADDF  @_const1,R2
      MPYF  R2,R9
      ADDF  R1,R9
      STF   R9,@_p_prime
*** 29      ----------------------    G = C$4 = 1.0F/(C$5-p_inf);
      LDF   @_p_inf,R1
      SUBF  R1,R9,R0
      CALL  INV_F40
      LDF   R0,R9
      STF   R0,@_G
```

```
*** 30      ----------------------    X = Y$1 = Xg[Y$0]*C$4-Xo[Y$0];
      LDA   @CONST+7,AR0
      MPYF  R0,*+AR0(IR1),R9
      LDA   @CONST+8,AR1
      SUBF  *+AR1(IR1),R9
      STF   R9,@_X
*** 31      ----------------------    *port1_out = Y$1;
      STF   R9,*AR5
*** 32      ----------------------    C$3 = i;
*** 32      ----------------------    T = Y$2 = Zg[C$3]*G-Zo[C$3];
      LDA   @CONST+9,AR1
      MPYF  R0,*+AR1(IR1),R9
      LDA   @CONST+10,AR1
      SUBF  *+AR1(IR1),R9
      STF   R9,@_T
*** 33      ----------------------    C$2 = j;
      LDA   @_j,IR1
*** 33      ----------------------    Y = Y$3 = si[C$2]*Y$2+Hy[C$2];
      LDA   @CONST+11,AR1
      MPYF  *+AR1(IR1),R9
      LDA   @CONST+12,AR1
      ADDF  *+AR1(IR1),R9
      STF   R9,@_Y
*** 34      ----------------------    *port1_out = Y$3;
      STF   R9,*AR5
*** 35      ----------------------    C$1 = j;
*** 35      ----------------------    Z = Y$4 = K$64[C$1]*T+K$69[C$1];
      LDF   @_T,R9
      MPYF  *+AR7(IR1),R9
      ADDF  *+AR6(IR1),R9
      STF   R9,@_Z
*** 36      ----------------------    *port1_out = Y$4;
      STF   R9,*AR5
*** 16      ----------------------    if ( (Y$5 = ++1) < 24u ) goto g2;
      LDI   @_1,R9
      ADDI  R9,1,R9
      STI   R9,@_1
      CMPI  24,R9
      BLO   L2
***   ----------------------    return;
EPI0_1:
      POP   AR7
      POP   AR6
      POP   AR5
      POP   AR4
      RETS
*************************************************************
* DEFINE CONSTANTS                                          *
*************************************************************
      .bss  CONST,13
      .sect ".cinit"
      .word 13,CONST
```

```
        .word       _co             ;0
        .word       _Hz             ;1
        .word       1048658         ;2
        .word       1048704         ;3
        .word       1048705         ;4
        .word       65527           ;5
        .float      4294967296.0    ;6
        .word       _Xg             ;7
        .word       _Xo             ;8
        .word       _Zg             ;9
        .word       _Zo             ;10
        .word       _si             ;11
        .word       _Hy             ;12
*******************************************************
* UNDEFINED REFERENCES                                 *
*******************************************************
        .globl      INV_F40
        .end
```

Annexe G : Algorithme FIT codé en assembleur et utilisant des LUTs

```
*********************************************************
*    Par : Eric Achard
*    Version : 27 novembre 1996
*    Changement :
*
*     10-10-96
*       - Insertion de la division dans le code
*       - Suppression des STORES inutiles
*       - Enlever les etapes de verification si negatif
*         lors de l'operation FLOAT
*
*   18-10-96
*     - Suppression de l'adressage indirect qui prend
*     trop de temps
*     - Rearangement des registres pour une meilleure
*     utilisation
*
*   25-10-96
*   - Paralleliser les operations de mul-acc
*
*   4-11-96
*       - Rajout du timer pour verifier la performance
*         du code [Note : pour obtenir le nombre de cycles,
*         faire (@100024H - 3) * 2 ]
*
*   27-11-96
*       - Enlever le calcul de p_prime
*
*   5-5-97
*       - Enlever la division pour mettre dans la LUT
*
*   8-5-97
*       - Enlever les tests sur le port de comm. 4
*
*********************************************************
*
      .version    40
FP    .set        AR3
      .globl      _Xg
      .globl      _Xo
      .globl      _Zg
    .globl  _Zo
      .globl      _Hy
      .globl      _Hz
      .globl      _si
      .globl      _co
    .globl  _G
    .globl  _main
```

```
*******************************************************
* FUNCTION DEF : _main
*******************************************************
_main:
*
***   ----------------------    Initialisation des tables
***   ----------------------    adresse de Xg[0]
    LDA     @CONST+0,AR0
***   ----------------------    adresse de Xo[0]
    LDA     @CONST+1,AR1
***   ----------------------    adresse de Zg[0]
    LDA     @CONST+2,AR2
***   ----------------------    adresse de Zo[0]
    LDA     @CONST+3,AR3
***   ----------------------    adresse de si[0]
    LDA     @CONST+4,AR4
***   ----------------------    adresse de Hy[0]
    LDA         @CONST+5,AR5
***   ----------------------    adresse de co[0]
    LDA     @CONST+6,AR6
***   ----------------------    adresse de Hz[0] dans R10
    LDI         @CONST+7,R10
*** ----------------------    adresse de G dans R9
    LDI     @CONST+13,R9
***   ----------------------    adresse de port1_out dans R5
    LDI         @CONST+8,R5
***   ----------------------    adresse de port4_sts
    LDA         @CONST+9,AR7
***   ----------------------    copie de l'adresse de port4_sts dans
R11
    LDI     AR7,R11
***   ----------------------    Initialisation des constantes
    LDI     0,R6
*** ----------------------    On fait un reset du timer dont le
registre
*** ----------------------    est a 100020H, donc port4_sts - 96
    LDI     03FFH,R1
    LDI     0FFFFFFFFH,R0
    STI     R0,*-AR7(88)
    STI     R1,*-AR7(96)
L2:
***   ----------------------    *port4_sts = *port4_sts&0xfff7;
    LDI     0FFF7H,R1
    AND     R1,*AR7,R0
    STI     R0,*AR7
L4:
***   ----------------------    p = *port4_in;
    LDA     *+AR7(1),IR0
***   ----------------------    j = *port4_in;
    LDI     *+AR7(1),R1
***   ----------------------    i = *port4_in;
    LDA     *+AR7(1),IR1
```

```
*** ----------------------    Debut du calcul
*** ----------------------    on met l'adresse de G dans AR7
    LDA     R9,AR7
*** ----------------------    On recupere G[p] dans R4
    LDF     *+AR7(IR0),R4
*** ----------------------    on met j dans IR0
    LDA     R1,IR0
*** ----------------------    Zg(i) * G -> R3
    MPYF3   *+AR2(IR1),R4,R3
*** ----------------------    Xg(i) * G -> R0 || Zg * G - Zo -> R2
    MPYF3   *+AR0(IR1),R4,R0
||  SUBF3   *+AR3(IR1),R3,R2
*** ----------------------    (Zg*G+Zo)*si(j) -> R1 || X -> R3
    MPYF3   *+AR4(IR0),R2,R1
||  SUBF3   *+AR1(IR1),R0,R3
*** ----------------------    (Zg*G+Zo)*co(j)->R0 || Y -> R2
    MPYF3   *+AR6(IR0),R2,R0
||  ADDF3   *+AR5(IR0),R1,R2
*** ----------------------    on met l'adresse de Hz dans AR7
    LDA     R10,AR7
*** ----------------------    Z -> R1
    ADDF3   *+AR7(IR0),R0,R1
***   ----------------------    on met l'adresse de port1_out dans AR7
    LDA     R5,AR7
***   ----------------------    *port1_out = X;
    STF     R3,*AR7
***     ----------------------    *port1_out = Y;
    STF     R2,*AR7
***   ----------------------    *port1_out = Z;
    STF     R1,*AR7
***     ----------------------    on met l'adresse de port4_sts dans
AR7
    LDA     R11,AR7
***   ----------------------    if ((1++ ) < 24 ) goto L4;
    ADDI    1,R6
    CMPI    1024,R6
    BLO     L4
*** ----------------------     On envoie la valeur du timer
    LDI     *-AR7(92),R0
*** ----------------------     Adresse de port1_out dans AR7
    LDA     R5,AR7
    STI     R0,*AR7
***   ----------------------    return;
EPI0_1:
      RETS
*********************************************************
*         CONSTANTS                                     *
*********************************************************
    .bss    CONST,14
      .sect ".cinit"
    .word   14,CONST
      .word _Xg         ;0
```

```
        .word _Xo          ;1
        .word _Zg          ;2
        .word _Zo          ;3
        .word _si          ;4
        .word _Hy          ;5
        .word       _co              ;6
        .word       _Hz              ;7
        .word       1048658          ;8
        .word       1048704          ;9
        .word       1048705          ;10
        .word       65527            ;11
        .float      4294967296.0     ;12
      .word   _G              ;13
*
        .end
```

Annexe H : Algorithme FIT codé en langage C pour être exécuté sur le DSP ADSP-21060

```
extern double Xg[], Xo[],Zg[],Zo[],si[],co[],Hy[],Hz[], p_inf;
unsigned int i, j, p, l;
double G, X, T, Y, Z;

void main(void)
{
      volatile int   *SRCTL0 = (int *)0x00E1;
      volatile int   *RX0    = (void *)0x00E3;
      volatile int   *RDIV0  = (int *)0x00E6;
      volatile int   *STCTL1 = (int *)0x00F0;
      volatile float *TX1    = (void *)0x00F2;
      volatile int   *TDIV1  = (int *)0x00F4;

    *RDIV0 = 0x00200001;
    *TDIV1 = 0x00200001;
    *SRCTL0 = 0x000065F1;       /* Ouvre le port de communication */
    *STCTL1 = 0x000065F1;       /* Ouvre le port de communication */

    for(l = 0;l < 24; l++)
    {
      /* attendre qu'une donnee soit arrivee*/
      while(!(*SRCTL0 & 0xc0000000));
            /* va lire p dans le port #0*/
            p = *RX0;

      /* attendre qu'une donnee soit arrivee*/
      while(!(*SRCTL0 & 0xc0000000));
            /* va lire j dans le port #0*/
            j = *RX0;

      /* attendre qu'une donnee soit arrivee*/
      while(!(*SRCTL0 & 0xc0000000));
            /* va lire i dans le port #0*/
            i = *RX0;

        G = 1 / (p - p_inf);
        X = Xg[i] * G - Xo[i];
        T = (G * Zg[i]) - Zo[i];
        Y = T * si[j] + Hy[j];
        Z = T * co[j] + Hz[j];
        *TX1 = X;
        *TX1 = Y;
        *TX1 = Z;
    }
}
```

Annexe I : Algorithme FIT codé en assembleur pour être exécuté sur le DSP ADSP-21060

```
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!     Par : Eric Achard
!     Version : 1
!     Date : 24 janvier 1997
!     Changement :
!
!     24-01-97
!           - Assembleur optimise.
!
!     21-02-97
!           - Parallelisation des lecture en memoire.
!
!     21-05-97
!           - mettre p dans la LUT.
!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

.segment /dm seg_dmda;
.gcc_compiled;
.endseg;


.segment /pm      seg_rth;
            NOP;
            JUMP _main;
.endseg;


.segment /pm seg_pmco;
.global     _main;
_main:

! Initialisation des registres
      m0=_Xg;
      m1=_Xo;
      m2=_Zg;
      m3=_Zo;
      m4=_si;
      m5=_Hy;
      m6=_co;
      m7=_p;
      R7=m7;
      R10=_Hz;
! Dans R15, on met la valeur du compteur de boucles
      R15=3;

!Initialisation des ports de communication.
      R0=0x0000C000;
```

```
      dm(0x00C7)=R0;
      R0=0X0003F03F;
      dm(0x00C8)=R0;
      R0=0x00009100;
      dm(0x00C6)=R0;
_valp:
! Attendre p
      i0=dm(0x00C2);

! Attendre j
      i1=dm(0x00C2);

! Attendre i
      i2=dm(0x00C2);

! DEBUT DES CALCULS
! On va chercher p dans la LUT
      m7=R7;
      f4=dm(m7,i0);
      m7=R10;
! Zg * G
      f8=f4*f0,f1=dm(m0,i2);
! Xg * G || Zg * G - Zo
      f9=f1*f4,f12=dm(m3,i2);
      f2=f8-f12,f5=dm(m4,i1);
! (Zg * G -Zo) * sin || Xg * G - Xo
      f8=f2*f5,f12=dm(m1,i2);
      f0=f9-f12,f6=dm(m6,i1);
! (Zg * G - Zo) * cos || [(Zg * G - Zo) * sin] + Hy
      f3=f2*f6,f12=dm(m5,i1);
      f1=f8+f12,f4=dm(m7,i1);
! [(Zg * G - Zo) * cos] + Hz
      f2=f3+f4;
! FIN DES CALCULS

! On ecrit X, Y et Z
      dm(0x00C3)=f0;
      dm(0x00C3)=f1;
      dm(0x00C3)=f2;
! On decremente de 1 le compteur de boucles.
      R0=1;
      R15=R15-R0;
! Si plus grand que 0, on recommence
      if eq jump (pc, _valp);

.extern     _p;
.extern     _Xg;
.extern     _Xo;
.extern     _Zg;
.extern     _Zo;
.extern     _si;
.extern     _Hy;
```

```
.extern     _co;
.extern     _Hz;

.endseg;
```

Annexe J : Coût de cartes disponibles sur le marché[1]

Cartes DSP

Nom de la compagnie	Produit	Bus	Processeur	Horloge MHz	Nb. De processeurs	Max. SRAM	Max. DRAM	Prix
Alacron Inc. (603) 891-2750	FT-SHARC	PCI,VME, ISA	ADSP-2106x	33,40	1,2,4,8	-	256Mx8	3,870$ US
Ariel Corp. (609) 860-2900	comm IO-Ip	VME	TMS320C40	48	-	-	512Mx8	3,995$ US
Bittware Research Systems (603) 226-0404	Blacktip	ISA	ADSP-21060	33,40	-	3.5Mx8	-	1,270$ US
	Snaggletooth	ISA	ADSP-21060	40	1 à 6	8Mx8	32Mx8	4,195$ US
	Megamouth	PCI	ADSP-21060	40	4	2Mx8	32Mx8	7,995$ US
Data Translation (800) 525-8528	Fulcrum	ISA	TMS320C40	40,50	-	1Mx8	32Mx8	2,000$ US
DSP Research (408) 773-1042	Tiger40/PC	ISA	TMS320C40	40	-	8Mx8	-	4,500$ US
	Tiger 440	ISA	TMS320C40	40	4	16Mx8	-	8,995$ US

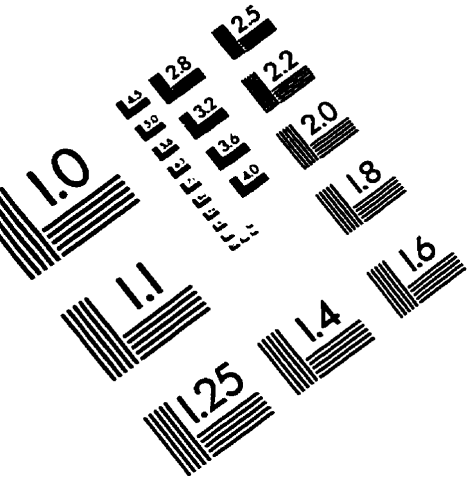
1 Les données suivantes proviennent du magazine Embedded Systems Programming ou sont directement disponible sur internet et d'autres de la compagnie elle-même.

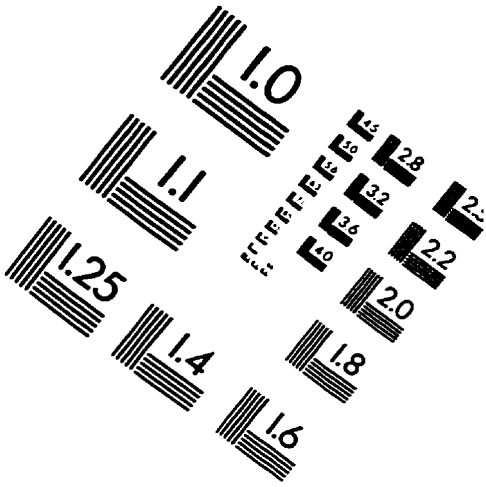
Nom de la compagnie	Produit	Bus	Processeur	Horloge MHz	Nb. De processeurs	Max. SRAM	Max. DRAM	Prix
Image & Signal Processing inc. (714) 970-0700	Blazer DSP Board	VME,VSB	TMS320C40	50	3 à 6	24Mx8	64Mx8	10,000$ US
Ixthos inc.	IXZ4/IXZ8	VME	ADSP-2106x	40	2,4,8	5Mx8	192Mx8	12,000$ US
Mercury Computer Systems (508) 256-1300	Race-MCH6 SHARC	VME	ADSP-21060	40	jusqu'à 12	6Mx8	256Mx8	38,100$ US
	Race-MCH9 SHARC	VME	ADSP-21060	40	jusqu'à 48	256Mx8	1024Mx8	51,200$ US
Pentek (201) 767-7100	4254 Single C40	VME	TMS320C40	50	-	4Mx8	-	4,995$ US
	4257 Dual C40	VME	TMS320C40	50	2	5Mx8	-	7,995$ US
	4269 Dual C40	VME	TMS320C40	50	2	8Mx8	-	6,995$ US
	4270 Quad C40	VME	TMS320C40	50	4	16Mx8	-	11,995$ US
	4284 Single C40	VME	TMS320C40	50	-	4Mx8	16Mx8	5,495$ US
	4285 Octal C40	VME	TMS320C40	50	jusqu'à 8	27Mx8	64Mx8	19,995$ US

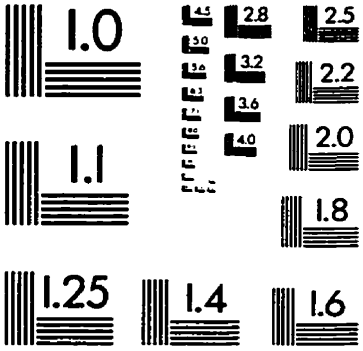
Cartes FPGA

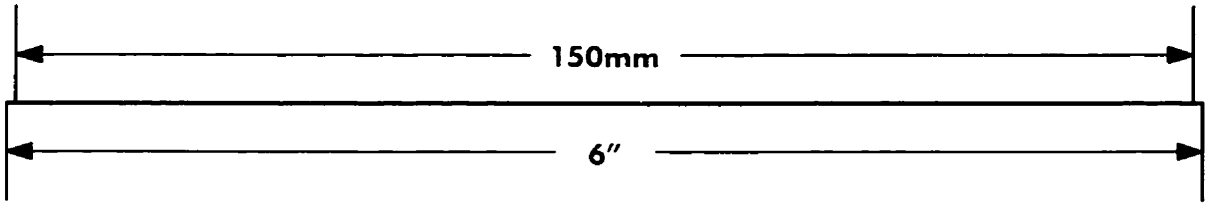
Nom de la compagnie	Produit	bus/comm	FPGA	nb. de FPGA	horloge MHz	max. SRAM	max. DRAM	prix
Mirotec (514) 956-0060	XT10-4MCP	Module TIM	XC4013	2	5 à 50	512k x 8	-	8,000$
	ZP10 support board	PCI	-	-	50	-	-	8,000$
	XC-436	Module TIM	XC4036EX	2	5 à 50	64k x 16	2M x8	9,000$
Giga Operations (510) 848-5446	XMOD	-	XC4010 XC4020 XC4028	2		256k x 8	8M x 8	900$ à 1,400$
	G900	PCI	-	-		-	-	1,400$

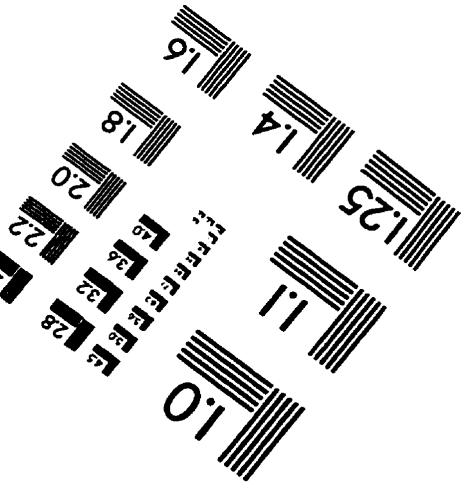
IMAGE EVALUATION
TEST TARGET (QA-3)

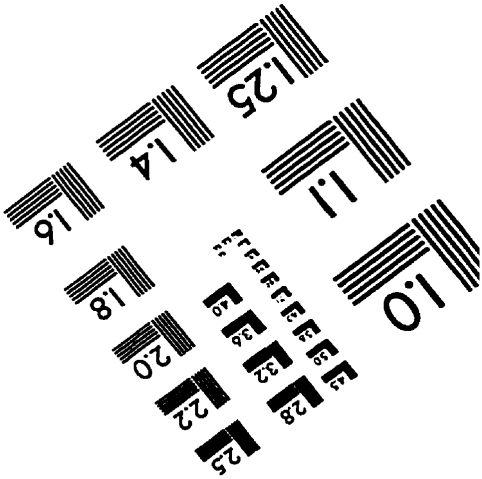












APPLIED IMAGE, Inc
1653 East Main Street
Rochester, NY 14609 USA
Phone: 716/482-0300
Fax: 716/288-5989